

# Universidad de Alcalá

## Escuela Politécnica Superior

Grado en Ingeniería en Electrónica y Automática  
Industrial

**Trabajo Fin de Grado**

Implementación de Algoritmos CORDIC con Vivado HLS

ESCUELA POLITECNICA

**Autor:** César Vázquez Alocén

**Tutor:** Ignacio Bravo Muñoz

2014



# UNIVERSIDAD DE ALCALÁ

ESCUELA POLITÉCNICA SUPERIOR

**Grado en Ingeniería en Electrónica y Automática Industrial**

**Trabajo Fin de Grado**

**Implementación de Algoritmos CORDIC con Vivado HLS**

Autor: César Vázquez Alocén

Director: Ignacio Bravo Muñoz

**Tribunal:**

**Presidente:** Pedro Martín Sánchez

**Vocal 1º:** Raúl Mateos Gil

**Vocal 2º:** Ignacio Bravo Muñoz

Calificación: .....

Fecha: .....



*A mis padres.*



Parte I

Resumen





# Resumen

En este trabajo se estudia la síntesis de alto nivel como metodología de diseño, para la implementación de algoritmos computacionalmente exigentes en plataformas de hardware reconfigurable (p.ej. FPGA). Para ello, se han elegido dos algoritmos extensamente documentados: el método de Jacobi para el cálculo de autovalores y autovectores; y el algoritmo CORDIC como elemento de cálculo del primero.

El objetivo principal, es implementar ambos algoritmos utilizando la herramienta Vivado HLS de Xilinx y comparar los resultados con los obtenidos mediante diseños equivalentes, uno codificado en VHDL y otro realizado mediante *Xilinx System Generator* (XSG).

Palabras clave: FPGA, Síntesis de alto nivel, Cálculo de autovalores y autovectores, CORDIC.



# Abstract

In this work we explore the use of High Level Synthesis (HLS) techniques, for the implementation of computationally expensive algorithms in reconfigurable hardware platforms like FPGA. For this task, two well known algorithms are used: the Jacobi method for eigenvalue and eigenvector calculation, and the CORDIC algorithm as main computational element.

The main goal is to implement both algorithms using the Vivado HLS tool from Xilinx and to compare them with equivalent designs developed using RTL methodologies (hand-coded VHDL and Xilinx System Generator).

Keywords: FPGA, High level synthesis, Eigenvalues and eigenvectors computation, CORDIC.



# Resumen extendido

En este trabajo se plantea el uso de la herramienta Vivado HLS de Xilinx (antiguo autoESL) para la implementación de algoritmos matemáticos de alta carga computacional. Este entorno de desarrollo se ubica dentro de una metodología de diseño hardware para plataformas reconfigurables (FPGA), que proporciona un mayor nivel de abstracción que el uso de lenguajes de descripción RTL<sup>1</sup> como VHDL.

El principal reclamo, es la posibilidad de utilizar lenguajes de programación de alto nivel para especificar los diseños de forma algorítmica (síntesis de alto nivel), generando de forma automática diseños RTL listos para ser utilizados en alguna otra herramienta de Xilinx. Estos sistemas pueden alcanzar diferentes grados de optimización (balance entre recursos consumidos y tiempo de ejecución), en función de una serie de directivas dadas por el usuario.

Puesto que la herramienta es relativamente nueva y la metodología de diseño adoptada, aunque ha estado en desarrollo durante muchos años, no ha tomado relevancia hasta ahora, junto con el trabajo se presenta una introducción tanto a Vivado HLS, como a la forma de diseñar mediante síntesis de alto nivel (apéndice A).

En el trabajo en si, en una primera parte se presenta la implementación de dos algoritmos matemáticos en Vivado HLS. El algoritmo CORDIC y el método de Jacobi para el cálculo de autovalores y autovectores en matrices simétricas, basado en el primero.

Se ha elegido el algoritmo de Jacobi por dos motivos principales:

- Está especialmente indicado para su uso en plataformas hardware, por el paralelismo inherente que presenta.
- Se dispone de implementaciones en FPGA (VHDL y Xilinx System Generator) que servirán para realizar una evaluación del sistema diseñado.

El problema de los autovalores se basa en dar solución a la ecuación:

$$V \cdot A = \lambda I \cdot V \quad (1)$$

Donde:

- $A \in \mathbb{R}^{n \times n}$  es una matriz real y simétrica.
- $V \in \mathbb{R}^{n \times n}$  es una matriz ortogonal que contiene los autovectores de  $A$ .
- $\lambda I \in \mathbb{R}^{n \times n}$  es una matriz diagonal que contiene los autovalores de  $A$ .

El algoritmo de Jacobi aborda el problema de forma iterativa, realizando una serie de transformaciones de semejanza del tipo  $R^T A R$  sobre la matriz inicial  $A$ , para hacerla más diagonal en cada iteración mediante la eliminación de los elementos de fuera de la diagonal.

---

<sup>1</sup>Register Transfer Level

Cuando lo que se plantea es una implementación hardware, existen diversas estrategias para minimizar la carga computacional y maximizar el paralelismo en cada iteración. Por un lado, el uso del algoritmo CORDIC para realizar las operaciones y, por otro lado, eliminar el mayor número de elementos fuera de la diagonal de  $A$  posible en cada iteración, realizando una división de la matriz  $A$  en submatrices de menor tamaño.

La expresión iterativa utilizada tiene la forma:

$$A^{(k+1)} = R(\alpha^{(k)})^T \cdot A^{(k)} \cdot R(\alpha^{(k)}) \quad (2)$$

Donde  $R$  es una matriz de rotación de Givens que, en el caso bidimensional se corresponde con:

$$R(\alpha) = \begin{bmatrix} \cos \alpha & -\sin \alpha \\ \sin \alpha & \cos \alpha \end{bmatrix} \quad (3)$$

El caso bidimensional es especialmente interesante, puesto que la división de la matriz inicial será en submatrices de dimensiones  $2 \times 2$ . Por ejemplo, si partimos de una matriz  $A \in \mathbb{R}^{n \times n}$ , el problema se abordará tratándola como una serie de  $n/2$  submatrices, sobre las que se aplicará la expresión (3), calculándose el ángulo  $\alpha$  a partir de los elementos de las submatrices diagonales, mediante una operación de *arcotangente*.

Teniendo en cuenta lo expuesto hasta ahora, en el método de Jacobi para el cálculo de autovectores y autovalores se presentan dos operaciones básicas: la operación de multiplicación de matrices  $R^T \cdot A \cdot R$  y el cálculo de arcotangentes. Ambas operaciones pueden ser resueltas mediante el algoritmo CORDIC, que permite resolver diferentes relaciones trigonométricas mediante operaciones de suma y desplazamiento.

Introducidos los fundamentos teóricos, se presenta una posible implementación de ambos algoritmos en Vivado HLS, utilizando el lenguaje de programación C y justificando los parámetros involucrados (número de iteraciones, codificación y ancho de palabra, tipos de datos a utilizar, ...) para alcanzar la solución más óptima posible en cuanto a consumo de recursos y error cometido en el cálculo.

La implementación se divide en dos partes: diseño y codificación del módulo CORDIC que sea capaz de resolver las operaciones involucradas en el algoritmo de Jacobi; e implementación del algoritmo de Jacobi a partir del módulo CORDIC.

Una vez codificados y validados ambos sistemas, en la segunda parte del trabajo se realizan diversas optimizaciones que llevan a alcanzar diferentes soluciones hardware sintetizables e implementables sobre FPGA. En todo momento se busca encontrar el equilibrio adecuado entre consumo de recursos y tiempo de ejecución.

Para llevar esto a cabo, se exploran diferentes alternativas, estudiando el consumo de recursos y la latencia en función del número de instancias hardware del algoritmo CORDIC y el tamaño de las matrices de entrada.

Una vez decidida la implementación adecuada, en la parte final del trabajo se analizan otras implementaciones tanto del algoritmo CORDIC como del método de Jacobi. En el caso del algoritmo CORDIC, se recurre a un sistema codificado en VHDL y a el *core* proporcionado por Xilinx.

En el caso del método de Jacobi, se toma un sistema que sigue la estrategia de división en submatrices  $2 \times 2$ , implementado de dos maneras diferentes. Por un lado, una versión codificada en VHDL y, por otro lado, un sistema igual al anterior pero diseñado mediante Xilinx System Generator (XGG).

A partir de los datos obtenidos del análisis de estos sistemas, se realiza una comparación con el módulo CORDIC y el sistema de cálculo de autovalores y autovectores diseñados, para evaluar el efecto que tiene en los resultados (error en el cálculo, tiempo de ejecución, frecuencia de reloj y recursos consumidos) el uso de síntesis de alto nivel en lugar de una metodología de diseño de bajo nivel.

# Índice general

<b>I</b>	<b>Resumen</b>	<b>vii</b>
	Resumen	ix
	Abstract	xi
	Resumen extendido	xiii
<b>II</b>	<b>Memoria</b>	<b>1</b>
<b>1.</b>	<b>Introducción</b>	<b>3</b>
1.1.	Presentación	3
1.2.	Objetivos y metodología	5
1.3.	Estructura del documento	6
<b>2.</b>	<b>Fundamentos Teóricos</b>	<b>7</b>
2.1.	Introducción y estado del arte	7
2.1.1.	Implementaciones hardware del algoritmo de Jacobi	8
2.2.	Algoritmo CORDIC	9
2.2.1.	Fundamentos matemáticos	9
2.2.2.	Modos de operación	11
2.2.3.	Región de convergencia	12
2.2.4.	Extensión del rango de convergencia	13
2.2.4.1.	Corrección de cuadrante en el modo rotación	14
2.2.4.2.	Corrección de cuadrante en el modo vectorización	15
2.3.	Algoritmo de Jacobi clásico	15
2.4.	Algoritmo de Jacobi por división en submatrices $2 \times 2$	17
2.4.1.	Utilización del algoritmo CORDIC para realizar las dobles rotaciones	19
<b>3.</b>	<b>Implementación en Vivado HLS</b>	<b>21</b>
3.1.	Implementación del algoritmo CORDIC	21
3.1.1.	Arquitecturas hardware	21
3.1.2.	Arquitectura serie	21
3.1.3.	Arquitectura paralela	22
3.1.4.	Codificación y ancho de palabra	22
3.1.5.	Implementación algorítmica	24
3.1.6.	Determinación del número óptimo de iteraciones	27
3.1.7.	Optimización del diseño	29
3.1.8.	Interfaz	31

3.2. Implementación del algoritmo de Jacobi . . . . .	32
3.2.1. Arquitecturas Hardware . . . . .	32
3.2.2. Implementación algorítmica . . . . .	34
3.2.3. Determinación del número óptimo de iteraciones . . . . .	35
3.2.4. Optimización del diseño . . . . .	37
3.2.5. Interfaz del módulo para el cálculo de autovalores y autovectores . . . . .	39
<b>4. Resultados y comparación con otras metodologías de diseño</b>	<b>43</b>
4.1. Algoritmo CORDIC . . . . .	43
4.1.1. <i>Core</i> de Xilinx . . . . .	43
4.1.2. Diseño en VHDL . . . . .	44
4.1.3. Comparación con el sistema diseñado mediante Vivado HLS . . . . .	45
4.2. Algoritmo de Jacobi . . . . .	47
4.2.1. Cálculo de autovalores y autovectores en XSG . . . . .	47
4.2.2. Cálculo de autovalores y autovectores codificado en VHDL sin <i>cores</i> . . . . .	47
4.2.3. Comparación con el sistema diseñado en Vivado HLS . . . . .	48
4.2.4. Comparación con la función <i>svd</i> de la librería de álgebra lineal de Vivado HLS . . . . .	49
<b>5. Conclusiones y trabajos futuros</b>	<b>51</b>
5.1. Conclusiones . . . . .	51
5.2. Trabajos futuros . . . . .	52
<b>III Pliego de condiciones</b>	<b>55</b>
<b>6. Pliego de condiciones</b>	<b>57</b>
6.1. Requisitos hardware . . . . .	57
6.2. Requisitos software . . . . .	57
<b>IV Presupuesto</b>	<b>59</b>
<b>7. Presupuesto</b>	<b>61</b>
<b>V Apéndices</b>	<b>63</b>
<b>A. Introducción a Vivado HLS</b>	<b>65</b>
A.1. Síntesis de alto nivel . . . . .	65
A.2. Instalación y entorno de trabajo . . . . .	66
A.3. Estructura de un diseño en Vivado HLS . . . . .	67
A.4. Especificación de interfaces . . . . .	68
A.4.1. Variables escalares . . . . .	69
A.4.2. Arrays y estructuras . . . . .	71
A.4.3. Interfaz AXI . . . . .	71
A.4.4. Especificación manual de interfaces . . . . .	71
A.5. Descripción de la funcionalidad y optimización . . . . .	71
A.5.1. Proceso de conversión . . . . .	72
A.5.2. Tipos de datos . . . . .	73
A.5.3. Operadores . . . . .	73
A.5.4. Bucles . . . . .	74
A.5.5. Arrays . . . . .	78
A.5.6. Optimización de la función completa . . . . .	81
<b>B. Manual de usuario</b>	<b>85</b>



B.1. Directorios del soporte informático . . . . .	85
B.2. Generación del módulo CORDIC . . . . .	85
B.3. Generación del módulo para el cálculo de autovalores y autovectores . . . . .	86
B.4. Simulación del algoritmo CORDIC y el método de Jacobi en Matlab . . . . .	86
<b>C. Código fuente de la implementación en Vivado HLS</b>	<b>87</b>
C.1. Módulo CORDIC . . . . .	87
C.1.1. <code>cordic_scp.c</code> . . . . .	87
C.1.2. <code>cordic_scp.h</code> . . . . .	89
C.2. Algoritmo de Jacobi . . . . .	90
C.2.1. <code>cordic_svd.c</code> . . . . .	90
C.2.2. <code>cordic_svd.h</code> . . . . .	93
<b>Bibliografía</b>	<b>96</b>



# Índice de figuras

1.1. Niveles de abstracción en el diseño digital . . . . .	4
1.2. Flujo de diseño en Vivado HLS (gráfico tomado del manual de Vivado HLS [1]) . . . . .	5
2.1. Funcionamiento del algoritmo CORDIC en coordenadas circulares . . . . .	11
2.2. Funcionalidades básicas del algoritmo CORDIC . . . . .	12
2.3. Rango de convergencia del algoritmo CORDIC, para los diferentes sistemas de coordenadas . . . . .	13
2.4. Diagrama de bloques del algoritmo CORDIC . . . . .	14
2.5. Reordenamiento interno y externo de los datos tras cada iteración . . . . .	18
3.1. Algoritmo CORDIC implementado mediante arquitectura serie . . . . .	22
3.2. Arquitectura paralela del algoritmo CORDIC . . . . .	23
3.3. Esquema de codificación XQN . . . . .	23
3.4. Esquemático del módulo DSP48E1 de la serie 7 de Xilinx . . . . .	24
3.5. Diagrama funcional del algoritmo CORDIC para N iteraciones . . . . .	25
3.6. Datos de entrada para validar el algoritmo CORDIC . . . . .	26
3.7. Error cometido en el modo rotación en función del número de iteraciones . . . . .	27
3.8. Error cometido en el modo vectorización en función del número de iteraciones . . . . .	28
3.9. Error máximo en ambos modos de funcionamiento en función del número de iteraciones . . . . .	28
3.10. Maquinas de estados generadas en el proceso de temporización del algoritmo CORDIC . . . . .	30
3.11. Interfaz RTL del módulo CORDIC diseñado . . . . .	32
3.12. Arquitectura sistólica para el cálculo de autovalores y autovectores en matrices simétricas de forma concurrente. . . . .	33
3.13. Arquitectura serie para el cálculo de autovalores y autovectores . . . . .	34
3.14. Diagrama de bloques funcional del algoritmo de Jacobi por división en matrices de $2 \times 2$ . . . . .	35
3.15. Reordenamiento simplificado de las filas y las columnas de la matriz tras cada iteración . . . . .	36
3.16. Error cometido en el cálculo de autovalores en función del número de iteraciones del algoritmo, para matrices de dimensiones $8 \times 8$ . . . . .	36
3.17. Error cometido en el cálculo de autovectores en función del número de iteraciones del algoritmo, para matrices de dimensiones $8 \times 8$ . . . . .	37
3.18. Latencia y recursos consumidos segmentando completamente el diseño, en función del número de módulos CORDIC, para matrices de dimensiones $8 \times 8$ . . . . .	38
3.19. Latencia y recursos consumidos segmentando el bucle principal, en función del número de módulos CORDIC, para matrices de dimensiones $8 \times 8$ . . . . .	39
3.20. Latencia y recursos consumidos segmentando el bucle principal, en función del orden de las matrices de entrada . . . . .	40
3.21. Interfaz RTL del algoritmo de Jacobi diseñado en Vivado HLS . . . . .	41

4.1. Comparación de recursos consumidos (slices) y latencia entre el módulo CORDIC diseñado en Vivado HLS y el sistema implementado mediante <i>cores</i> de Xilinx . . . . .	46
4.2. Comparativa de latencia y frecuencia de reloj máxima para el algoritmo de Jacobi implementado mediante diferentes metodologías de diseño. . . . .	48
4.3. Error porcentual máximo cometido en el cálculo de autovalores y autovectores por los tres sistemas bajo análisis. . . . .	49
4.4. Resultados obtenidos tras la síntesis en Vivado HLS de la función <i>svd</i> de la librería de álgebra lineal de Xilinx para matrices de dimensiones $8 \times 8$ . . . . .	50
4.5. Comparación del consumo de recursos entre la función <i>svd</i> de la biblioteca de álgebra lineal de Vivado HLS y el sistema con <i>pipeline</i> en el bucle principal y cuatro módulos CORDIC en modo rotación, para matrices de dimensiones $8 \times 8$ . . . . .	50
A.1. Perspectiva de Síntesis del entorno Vivado HLS . . . . .	67
A.2. Perspectiva de Análisis del entorno Vivado HLS . . . . .	68
A.3. Puertos generados en la síntesis de la función <i>sum_io</i> . . . . .	70
A.4. Fases en el proceso de síntesis realizado por Vivado HLS . . . . .	72
A.5. Análisis del multiplicador de matrices sin optimizar . . . . .	75
A.6. Análisis del multiplicador de matrices tras varias optimizaciones . . . . .	77
A.7. Accesos a memoria del multiplicador de matrices tras diferentes optimizaciones . . . . .	77
A.8. Interfaz del multiplicador de matrices segmentado . . . . .	79
A.9. Implementación del multiplicador de matrices con Pipeline + Reshape . . . . .	80

# Índice de tablas

3.1. Resultados de la síntesis del módulo CORDIC en Vivado HLS, sin aplicar ninguna optimización . . . . .	27
3.2. Resultados de la síntesis en Vivado HLS sin optimizar y con <i>pipeline</i> en el bucle principal. . . . .	29
3.3. Resultados de la síntesis en Vivado HLS de CORDIC deshaciendo el bucle principal . . . . .	31
3.4. Resultados de la síntesis en Vivado HLS del algoritmo de Jacobi, sin realizar ninguna optimización . . . . .	35
3.5. Diseño segmentado totalmente para matrices de entrada de dimensiones $8 \times 8$ , sintetizado para el dispositivo xc7a100tcsg324-1 . . . . .	38
4.1. Señales en los módulos CORDIC proporcionados por Xilinx . . . . .	44
4.2. Resultados de la implementación de los tres diseños sobre el dispositivo xc7a100csg324-1 (Artix 7) . . . . .	46
4.3. Resultados de la implementación del sistema realizado mediante diferentes metodologías de diseño, en el dispositivo xc7a100tcsg324-1. . . . .	48
7.1. Coste de los recursos software . . . . .	61
7.2. Coste de los recursos hardware . . . . .	61
7.3. Coste de la mano de obra . . . . .	61
7.4. Coste del material fungible y los libros . . . . .	62
A.1. Latencia del bucle y del diseño en el multiplicador de matrices con Pipeline + Reshape . . . . .	80
A.2. Latencia de las diferentes alternativas de optimización y consumo de recursos . . . . .	81
A.3. Consumo de recursos y latencia en el multiplicador de tres matrices, para diferentes alternativas de optimización . . . . .	82



Parte II

Memoria





# Capítulo 1

## Introducción

En este capítulo introductorio se expone el contexto en el que se desarrolla el trabajo, los principales objetivos que se han perseguido y la metodología de trabajo seguida. Finalmente, se presenta la estructura del resto del documento para una mejor comprensión.

### 1.1. Presentación

Cuando se requiere implementar algoritmos de alta carga computacional en plataformas de procesamiento tradicionales (p.ej.  $\mu P$ ), pueden surgir diversos problemas si la complejidad de los cálculos es elevada, derivados principalmente de los cuellos de botella que genera una arquitectura hardware no adaptada a la aplicación (unidades de cálculo, arquitectura de la memoria, ...).

Cuando los requisitos de la aplicación no pueden ser cubiertos por este tipo de sistemas, se plantean diferentes posibilidades. En ocasiones, es suficiente el uso de procesadores de propósito específico como los DSP<sup>1</sup>, que incluyen periféricos de uso común en determinadas aplicaciones como el procesamiento de señal, la visión artificial o la electrónica de potencia (filtros digitales, unidades MAC, generadores PWM, ...).

En el siguiente escalón encontramos soluciones puramente hardware, como ASIC<sup>2</sup> o FPGA<sup>3</sup>. Salvo en ocasiones donde se plantea la fabricación de muchas unidades, la primera opción queda descartada por los elevados tiempos de desarrollo y altos costes que conlleva, siendo el uso de FPGAs la alternativa más popular para implementar sistemas hardware a medida.

La ventaja fundamental de las FPGA es el paralelismo inherente que presentan, siendo su mayor inconveniente la dificultad de aprovechar este paralelismo en determinadas aplicaciones y el bajo nivel al que ha de trabajarse para conseguir sistemas eficientes. De forma resumida, cuando se diseña de FPGA los pasos fundamentales son:

1. Establecimiento de las especificaciones de la aplicación.
2. Prueba de los algoritmos a nivel de software mediante lenguajes de programación de alto nivel como C/C++ o MATLAB.
3. Estudio de las dependencias de datos en el algoritmo, división en tareas y estudio de la carga computacional de cada módulo.

---

<sup>1</sup>Digital Signal Processor

<sup>2</sup>Application-Specific Integrated Circuit

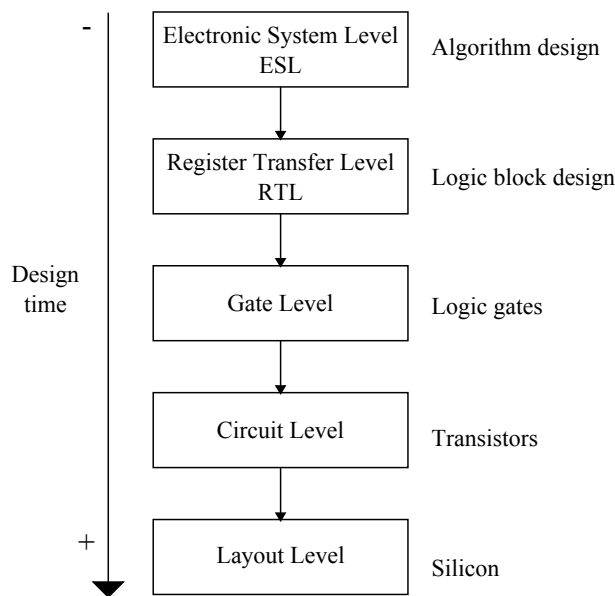
<sup>3</sup>Field Programmable Gate Array

4. Definición de una arquitectura hardware e implementación a bajo nivel mediante lenguajes RTL, como VHDL, junto con la lógica de control del sistema.
5. Verificación de la correcta funcionalidad del diseño.
6. Síntesis e implementación sobre un determinado dispositivo.

En esta forma de trabajar podemos señalar dos puntos que consumen la mayor parte del tiempo de diseño. Por un lado, la paralelización de los algoritmos puede ser, en ocasiones, un proceso laborioso y sujeto a errores y, por otro lado, la descripción RTL es un proceso poco productivo por el bajo nivel al que se trabaja.

Con el objetivo de incrementar la velocidad a la hora de diseñar, constantemente surgen herramientas que fomentan la reutilización de módulos previamente diseñados, facilitando su integración. Una de las más conocidas es XSG<sup>4</sup> de Xilinx, que permite implementar sistemas de forma gráfica en Simulink, combinando elementos discretos y *cores* propietarios. Además, permite agilizar el proceso de verificación funcional, pudiéndose realizar dentro de MATLAB.

Sin embargo, estas nuevas formas de diseñar no resuelven el inconveniente de la conversión de un algoritmo especificado en software a hardware. La solución a este problema la encontramos en el paradigma de diseño ESL<sup>5</sup>. Si se observa la figura 1.1, pueden verse las diferentes metodologías de diseño hardware ordenadas de mayor a menor nivel de abstracción.



**Figura 1.1:** Niveles de abstracción en el diseño digital

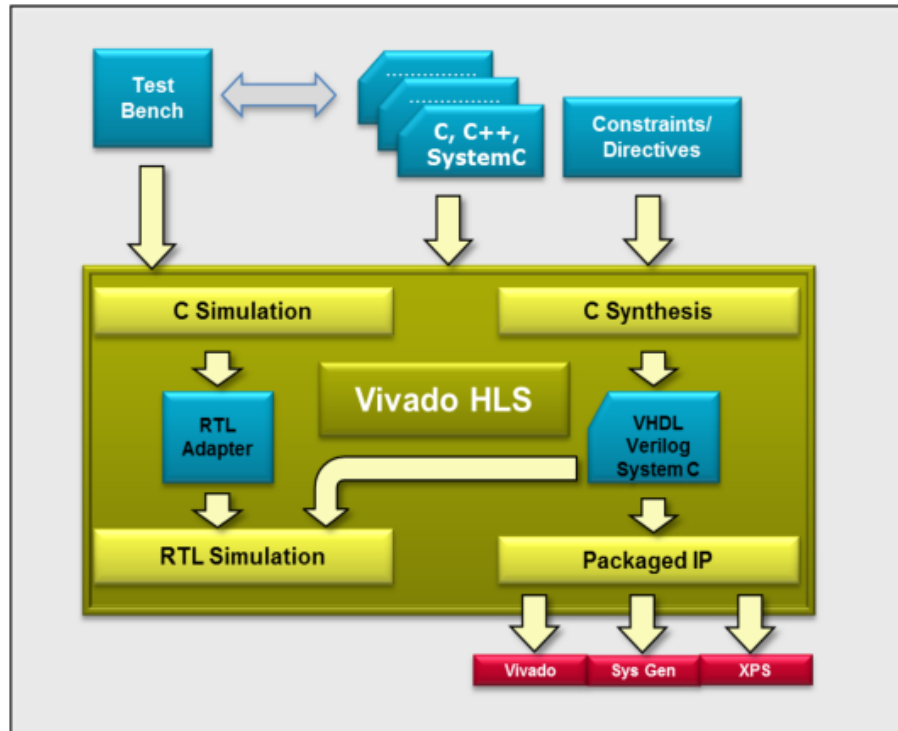
El objetivo de la metodología ESL y de las herramientas que la siguen, es permitir la especificación funcional de los algoritmos, siendo el diseño del hardware de bajo nivel y la generación de la lógica de control transparente al diseñador, que solo debe proporcionar directivas de optimización para indicar restricciones físicas y temporales (recursos consumidos, frecuencia de reloj, ...).

Dentro de este paradigma encontramos las herramientas de síntesis de alto nivel (HLS<sup>6</sup>), que permiten describir un sistema de forma funcional, utilizando lenguajes de programación de alto nivel como Python o C/C++. Aunque estos entornos de desarrollo han existido durante muchos años, la mayoría eran de propósito específico, o estaban limitados a diseños pequeños y con características muy concretas.

<sup>4</sup>Xilinx System Generator

<sup>5</sup>Electronic System Level [design]

<sup>6</sup>High Level Synthesis



**Figura 1.2:** Flujo de diseño en Vivado HLS (gráfico tomado del manual de Vivado HLS [1])

Es ahora cuando han surgido herramientas que permiten la realización de diseños relativamente complejos en software y su conversión a sistemas RTL, por ejemplo, Impulse C y Vivado HLS. En este trabajo se ha utilizado la herramienta Vivado HLS para implementar el algoritmo de Jacobi para el cálculo de autovalores y autovectores, que destaca por su alta carga computacional.

La forma de trabajar difiere de la ya comentada para las metodologías de diseño tradicionales, pudiéndose resumir en los siguientes puntos:

1. Depuración y validación software de un diseño en C/C++.
2. Generación de un sistema hardware de igual funcionalidad y optimización iterativa mediante directivas.
3. Simulación del hardware generado para verificar que se ha capturado correctamente la funcionalidad.
4. Integración en el flujo de desarrollo RTL del sistema generado, exportando el hardware como módulo VHDL, PCore para EDK<sup>7</sup>, o bloque de XSG.

## 1.2. Objetivos y metodología

Como se ha comentado, en este trabajo se trabaja con Vivado HLS para implementar el algoritmo de Jacobi sobre hardware reconfigurable, con el objetivo de determinar si las herramientas de síntesis de alto nivel tienen un nivel de madurez suficiente, para integrarlas en el flujo de desarrollo RTL.

Hay varios motivos que justifican la elección de este algoritmo. El cálculo de autovalores y autovectores es muy utilizado en aplicaciones que requieren el procesamiento de datos en tiempo real, siendo en

<sup>7</sup>Embedded Development Kit

ocasiones necesaria una implementación hardware para poder cumplir con las restricciones temporales.

Además, se dispone de diseños para FPGA del sistema, realizados utilizando metodologías RTL (VHDL y XSG), que sirven como base para evaluar las prestaciones del sistema realizado. Para este último punto, se han utilizado criterios de:

- Tiempo de desarrollo.
- Error cometido en el cálculo.
- Recursos internos de la FPGA empleados.
- Tiempo de ejecución y frecuencia máxima de reloj alcanzada.

El punto de partida del trabajo, es la arquitectura sistólica presentada en [2] por Brentt y Luk, para el cálculo de los valores singulares de una matriz  $A \in \mathbb{R}^{n \times n}$  mediante su descomposición en submatrices de dimensiones  $2 \times 2$ , utilizando el método iterativo de Jacobi. Además, como elemento principal de cálculo se ha utilizado el algoritmo CORDIC, siguiendo la propuesta de Cavallaro [3].

Esto implica que el primer paso ha sido la especificación software de la arquitectura, para facilitar su implementación en lenguaje C con Vivado HLS. El cuerpo del trabajo puede dividirse en tres partes:

- Estudio teórico y simulación del algoritmo en MATLAB para determinar el tipo de codificación adecuado.
- Implementación y optimización del algoritmo CORDIC en Vivado HLS.
- Uso del módulo CORDIC diseñado para implementación del algoritmo de Jacobi.

La descripción del módulo CORDIC permite introducir el uso de Vivado HLS, y la forma de proceder a la hora de optimizar de forma iterativa los diseños, para determinar el tipo de arquitectura hardware que mejor se adapta a los requisitos de la aplicación (paralela, serie, segmentada, ...), siguiendo una metodología similar en el resto del diseño.

### 1.3. Estructura del documento

Para concluir, se describe la estructura del resto del documento. En el capítulo 2 se presenta en primer lugar el problema de los autovalores y los métodos más importantes utilizados para su determinación numérica. A continuación, se realiza el desarrollo teórico, tanto del algoritmo CORDIC, como del método de Jacobi para el cálculo de autovalores y autovectores.

En el capítulo 3 se presenta la implementación y optimización de ambos algoritmos utilizando la herramienta Vivado HLS de Xilinx, justificando el valor de los parámetros seleccionados (tipo de codificación y ancho de palabra, numero de iteraciones, ...).

Una vez completado el desarrollo del sistema, en el capítulo 4 se comparan los resultados obtenidos con los de diseños de igual funcionalidad, realizados utilizando VHDL (sin uso de *cores* propietarios) y XSG.

A continuación, en el capítulo 5 se presentan las conclusiones a las que ha llevado trabajar con síntesis de alto nivel como alternativa a otras formas de diseñar, y los trabajos futuros que podrían desarrollarse como continuación.

Finalmente, en los capítulos 6 y 7 se presentan en pliego de condiciones y el presupuesto. Además, se ha incluido una serie de apéndices, donde encontramos una introducción a Vivado HLS (apéndice A), el manual de instrucciones (apéndice B), que detalla como poner en marcha los diseños desarrollados y reproducir los resultados, y el código fuente (apéndice C) de la implementación en Vivado HLS.

# Fundamentos Teóricos

## 2.1. Introducción y estado del arte

El cálculo de autovalores y autovectores presenta numerosas aplicaciones en la ciencia y la ingeniería. Dentro de ellas podemos distinguir aquellas en las que no es necesaria su obtención en tiempo real (simulaciones), como por ejemplo el análisis de vibraciones en estructuras o la física cuántica, y las que requieren el cálculo en tiempo real como parte de un proceso de toma de decisiones, por ejemplo la técnica estadística PCA<sup>1</sup> (visión artificial, procesamiento de señal, control de procesos, ...).

En el primer caso, se plantean matrices con tamaños del orden de  $10^4 \times 10^4$  elementos, y se requiere una gran precisión en el cálculo mientras que, en el segundo caso, las matrices presentan tamaños uno o dos órdenes de magnitud menores, dependiendo de la aplicación.

De forma analítica, los autovalores de una matriz  $A \in \mathbb{C}^{n \times n}$  se corresponden con las  $n$  raíces de su polinomio característico (2.1).

$$p(\lambda) = \det(\lambda I - A) \tag{2.1}$$

Por otro lado, los autovectores correspondientes a los autovalores de la matriz  $A$ , son los vectores no nulos tales que:

$$A\vec{v} = \lambda\vec{v} \tag{2.2}$$

Puesto que el tamaño de las matrices a analizar suele tener un orden mayor que cuatro, el método analítico no es válido (debido a la imposibilidad de calcular las raíces del polinomio), siendo necesario recurrir a métodos iterativos para resolver el problema.

Dentro de los métodos iterativos aparecen dos grupos: los que realizan una transformación inicial sobre la matriz de entrada, convirtiéndola a un formato más manejable e iterando sobre la matriz transformada y los que trabajan directamente con ella.

Dentro del primer grupo, el método más extendido es la descomposición QR [4], en la que se suele partir de una matriz tridiagonal. En el segundo grupo, podemos destacar el algoritmo de Jacobi, el más utilizado en sus distintas variantes, especialmente para el caso de matrices cuadradas y simétricas.

---

<sup>1</sup>Principal Component Analysis

### 2.1.1. Implementaciones hardware del algoritmo de Jacobi

Aunque el algoritmo de Jacobi data de mediados del siglo XIX, no se popularizó hasta finales del siglo XX. El motivo principal es que, a partir de tamaños de matrices de  $10 \times 10$ , la carga computacional es muy elevada en comparación con otros métodos de cálculo como la descomposición QR.

La ventaja principal que presenta el método de Jacobi, es que presenta un error muy reducido, esto ha propiciado numerosas investigaciones para acelerar su ejecución, que buscan paralelizar las operaciones.

El objetivo fundamental del algoritmo de Jacobi, es diagonalizar una matriz de forma iterativa, eliminando un elemento fuera de la diagonal en cada iteración. Para ello, se vale de una serie de transformaciones unitarias, basadas en la matriz de rotación de Givens (2.3).

$$R(\theta) = \begin{bmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & C_\theta & \cdots & S_\theta & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & -S_\theta & \cdots & C_\theta & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{bmatrix} \quad (2.3)$$

Estudiando las dependencias de datos en cada iteración, se llegó a la conclusión de que es posible eliminar  $m$  elementos en cada iteración de manera simultánea, dividiendo la matriz de entrada en  $m$  submatrices. Esta variante del algoritmo se denomina *block Jacobi method* [4].

Partiendo de esta versión modificada, Brent y Luk proponen en [5] una arquitectura hardware para realizar el cálculo, dividiendo una determinada matriz de entrada  $A \in \mathbb{R}^{n \times n}$ , en  $\frac{n}{2} \times \frac{n}{2}$  submatrices de dimensiones  $2 \times 2$ .

Además, en [3] Cavallaro desarrolló un método que permite utilizar el algoritmo CORDIC para realizar todas las operaciones del método de Jacobi, siendo necesario añadir únicamente lógica de control. La desventaja principal de esta arquitectura, es el elevado número de recursos consumidos, que crece a razón de  $O(n^2)$  con el orden  $n$  de las matrices de entrada.

Con este problema en mente, en [6] se propone un sistema basado en los anteriores, que hace uso de un único módulo de cálculo, pero manteniendo la metodología de la división en submatrices, demostrando que es posible conseguir contener el consumo de recursos sin incrementar de forma equivalente el tiempo de ejecución.

Por último, en [7] se implementa el sistema utilizando la herramienta XSG de Xilinx. Este trabajo y el anterior sirven como punto de partida para el sistema presentado en este proyecto y como base para la evaluación de los resultados logrados mediante Vivado HLS.

En el resto del capítulo, se presenta en primer lugar el algoritmo CORDIC, puesto que es la base del método de Jacobi para el cálculo de autovalores y autovectores.

En segundo lugar, se desarrolla el algoritmo de Jacobi y se justifica el uso de CORDIC para realizar todas las operaciones involucradas.

## 2.2. Algoritmo CORDIC

El algoritmo CORDIC<sup>2</sup> fue presentado en 1959 por Jack E. Volder [8] como un método de cálculo en tiempo real de relaciones trigonométricas, en sistemas de navegación aérea.

Posteriormente, en 1972 J. S. Walther [9] generalizó el algoritmo para su uso en el cálculo de multitud de funciones elementales: multiplicación, división, relaciones trigonométricas, raíces cuadradas, etc.

La ventaja fundamental del algoritmo CORDIC radica en que su implementación más básica, solo requiere operaciones de suma y de desplazamiento de bits. Esto lo hace especialmente adecuado para su implementación sobre hardware reconfigurable, donde los multiplicadores son un recurso escaso.

De forma intuitiva, el funcionamiento consiste en la rotación de un vector de entrada un determinado ángulo, para obtener otro vector a la salida.

Dependiendo del sistema de coordenadas en el que se trabaje (lineal, circular o hiperbólico), es posible realizar el cálculo de diferentes funciones elementales a partir de la relación entre ambos vectores.

En este trabajo el módulo CORDIC deberá ser capaz de trabajar en coordenadas circulares, realizando dos operaciones. Por un lado el cálculo de arcotangentes (modo vectorización) y, por otro lado, rotaciones de un vector de entrada  $\vec{v}$  un ángulo dado  $\alpha$ .

### 2.2.1. Fundamentos matemáticos

Sea un vector  $\vec{v}_i = (x_i, y_i)$  dado, podemos expresar su módulo y fase como:

$$|\vec{v}| = [x^2 + my^2]^{1/2} \quad (2.4)$$

$$\angle \vec{v} = m^{-1/2} \tan^{-1} \left( m^{1/2} \frac{y}{x} \right) \quad (2.5)$$

Donde  $m = \{-1, 0, 1\}$  es un parámetro que determina el sistema de coordenadas en el que se trabaja:

- Si  $m = 0$  el vector está expresado en coordenadas lineales.
- Si  $m = 1$  el vector está expresado en coordenadas circulares.
- Si  $m = -1$  el vector está expresado en coordenadas hiperbólicas.

Como se ha comentado, el objetivo es rotar el vector de entrada  $\vec{v}_i$  un determinado ángulo  $\alpha$ . En lugar de hacerlo en una única rotación, se hace de forma iterativa, rotando un ángulo de  $\alpha_{i,m}$  cada vez denominado micro-rotación.

$$\alpha_{i,m} = m^{-1/2} \cdot \tan^{-1} (\sqrt{m} \cdot \delta_i) \quad (2.6)$$

El ángulo total a rotar puede expresarse como la suma de todas las micro-rotaciones realizadas (2.7).

$$\alpha = \sum_i \pm \alpha_{i,m} \quad (2.7)$$

Particularizando para el sistema de coordenadas circulares, si expresamos la rotación de un vector  $\vec{v}_i = (x_i, y_i)$  un ángulo  $\alpha_i$ , obtendremos un nuevo vector  $\vec{v}_{i+1} = (x_{i+1}, y_{i+1})$ , cuyas coordenadas vienen dadas por:

$$\begin{aligned} x_{i+1} &= x_i \cos \alpha_i - y_i \sin \alpha_i \\ y_{i+1} &= y_i \cos \alpha_i + x_i \sin \alpha_i \end{aligned} \quad (2.8)$$

---

<sup>2</sup>COordinate Rotation DIgital Computer

En la introducción, se mencionó que el algoritmo CORDIC puede implementarse mediante únicamente operaciones de suma y desplazamiento, luego el siguiente paso es eliminar los productos y relaciones trigonométricas de la expresión 2.8. Para ello, en primer lugar sacamos factor común a  $\cos \alpha_i$ , llegando a un nuevo sistema de ecuaciones (2.9).

$$\begin{aligned} x_{i+1} &= \cos \alpha_i (x_i - y_i \tan \alpha_i) \\ y_{i+1} &= \cos \alpha_i (y_i + x_i \tan \alpha_i) \end{aligned} \quad (2.9)$$

Tomando el valor de  $\alpha_i$  de la ecuación (2.6) tenemos que, en coordenadas circulares,  $\tan \alpha_i = \delta_i$ . Sustituyendo  $\tan \alpha_i$  en la expresión 2.10, obtenemos un nuevo sistema de ecuaciones dado por:

$$\begin{aligned} x_{i+1} &= \cos \alpha_i (x_i - y_i \delta_i) \\ y_{i+1} &= \cos \alpha_i (y_i + x_i \delta_i) \end{aligned} \quad (2.10)$$

Llegados a este punto, podemos seleccionar un valor de  $\delta_i$  que permita simplificar la operación:

$$\delta_i = \sigma_i 2^{-i} \quad (2.11)$$

Donde  $\sigma \in \{-1, 1\}$  indica el sentido de la rotación. Trabajar con potencias de dos, permite realizar multiplicaciones y divisiones mediante desplazamientos de bits, reduciendo el número de recursos consumidos. Si sustituimos  $\delta_i$  por su nuevo valor y eliminamos  $\cos \alpha_i$  de las ecuaciones se llega a:

$$\begin{aligned} x_{i+1} &= x_i - y_i \sigma_i 2^{-i} \\ y_{i+1} &= y_i + x_i \sigma_i 2^{-i} \\ z_{i+1} &= z_i - \sigma_i \alpha_i \end{aligned} \quad (2.12)$$

Notesé que en (2.12) se ha añadido una nueva variable  $z$  que acumula el valor del ángulo rotado. Estas son las denominadas *ecuaciones hardware* del algoritmo CORDIC, que implementan la funcionalidad principal. Si generalizamos la expresión (2.12) para cualquier sistema de coordenadas [10], llegamos finalmente a:

$$\begin{aligned} x_{i+1} &= x_i - y_i m \sigma_i 2^{-i} \\ y_{i+1} &= y_i + x_i \sigma_i 2^{-i} \\ z_{i+1} &= z_i - \sigma_i \alpha_{i,m} \end{aligned} \quad (2.13)$$

Puesto que se ha eliminado  $\cos \alpha_i$  de las ecuaciones, cada vez que se realiza una rotación el módulo del vector se verá modificado por un determinado factor de escala  $K_{i,m}$  (2.14), como puede observarse en la figura 2.1a.

$$K_{i,m} = \frac{1}{\cos(\sigma_i m^{1/2} \alpha_i)} = \sqrt{1 + m \sigma_i^2 2^{-2i}} \quad (2.14)$$

Tras  $n$  micro-rotaciones, el factor de escala tomará un valor de

$$K_m = \prod_{i=0}^{n-1} \sqrt{1 + \sigma_i^2 \cdot 2^{-2i}} \quad (2.15)$$

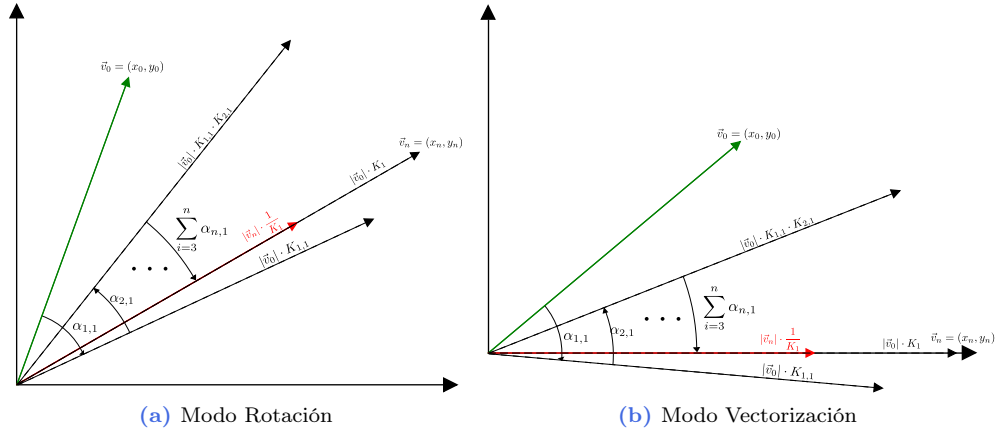
En la ecuación 2.15 encontramos que el aumento de módulo del vector, no depende del sentido en el que se realice la rotación, puesto que  $\sigma_i^2 \in \{-1, 1\}^2 = 1$ . Esto implica que  $K_m$  es conocido a priori, pudiendo calcularse *offline* y corregirse tras concluir todas las micro-rotaciones.

La forma más sencilla de corregir el factor de escala, es calcular previamente su valor inverso  $1/K_{i,m}$ , y multiplicar por el las variables tras finalizar la operación. Dependiendo del tipo de implementación elegido, es posible utilizar otros métodos, como la introducción de nuevas micro-rotaciones o la multiplicación por los factores de escala parciales ( $K_{i,m}$ ).



### 2.2.2. Modos de operación

Hasta ahora se ha presentado un desarrollo genérico del algoritmo de la rotación iterativa de un vector cualquiera  $\vec{v}_i = (x_i, y_i)$ . En general, el objetivo de las micro-rotaciones es aproximar a cero la variable  $z$  o la variable  $y$ , determinando esto el resultado obtenido.



**Figura 2.1:** Funcionamiento del algoritmo CORDIC en coordenadas circulares

- Cuando se busca hacer cero la variable  $z$ , se está trabajando en modo *rotación*. En coordenadas circulares y modo rotación, tras  $n$  micro-rotaciones se obtendrán las coordenadas del vector  $\vec{v}_i$ , rotado un ángulo  $z_i$ .
- En modo *vectorización* se trata de hacer cero la variable  $y$ . Al finalizar las  $n$  micro-rotaciones, idealmente el vector se encontrará sobre el eje de coordenadas. En otras palabras, la variable  $x$  contendrá el módulo de  $\vec{v}_i$ , y en  $z$  se habrá acumulado el ángulo total rotado, es decir, la fase de  $\vec{v}_i$ .

En la figura 2.1a se muestra el funcionamiento en el modo *rotación*. Se parte de un vector  $\vec{v}_0$  (verde), que se pretende rotar un ángulo  $\alpha$  hasta alcanzar la posición de  $\vec{v}_n$ . Para ello, se va aproximando mediante micro-rotaciones, hasta que los vectores coincidan ( $z = 0$ ).

Por otro lado, en la figura 2.1b se ilustra el funcionamiento en el modo vectorización. En este caso, se busca hacer cero la variable  $y$ , es decir, situar el vector inicial  $\vec{v}_0$  sobre el eje de coordenadas. Para ello, se realizan micro-rotaciones sobre  $\vec{v}_0$  aproximando su componente vertical a cero de forma sucesiva.

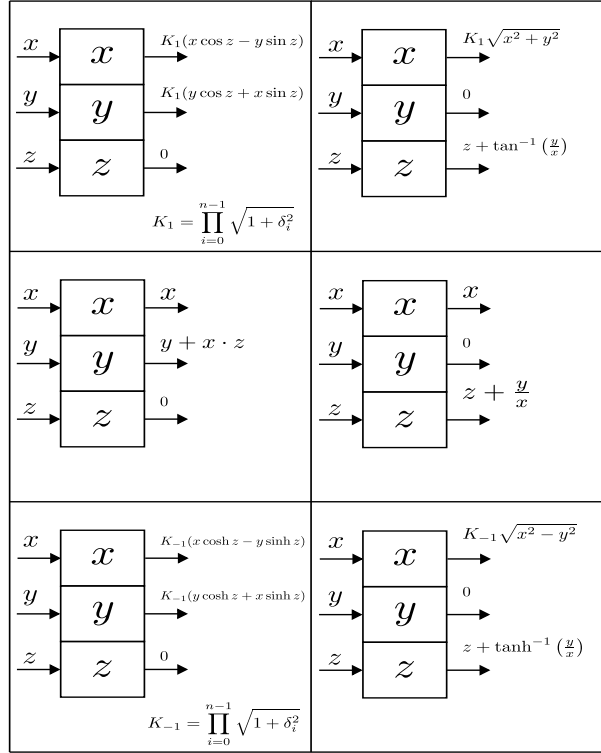
De forma general, tras  $n$  micro-rotaciones las ecuaciones tomarán la forma:

$$\begin{aligned} x_n &= K_m \cdot \prod_{i=0}^{n-1} (x_i - m\sigma_i 2^{-i} y_i) \\ y_n &= K_m \cdot \prod_{i=0}^{n-1} (y_i + \sigma_i 2^{-i} x_i) \\ z_n &= z_i - \alpha \end{aligned} \tag{2.16}$$

Si se particularizan estas expresiones para cada sistema de coordenadas y ambos modos de funcionamiento, las relaciones entre la entrada y la salida son las mostradas en la figura 2.2.

Además de determinar las expresiones de salida, la forma de elegir el parámetro  $\sigma_i$  también viene determinada por el modo de operación. En el caso del modo *rotación*, el valor de  $\sigma_i$  vendrá dado por:

$$\sigma_i = \text{sign}(z_i) \tag{2.17}$$



**Figura 2.2:** Funcionalidades básicas del algoritmo CORDIC

Mientras que, en el modo *vectorización*:

$$\sigma_i = -\text{sign}(y_i) \quad (2.18)$$

### 2.2.3. Región de convergencia

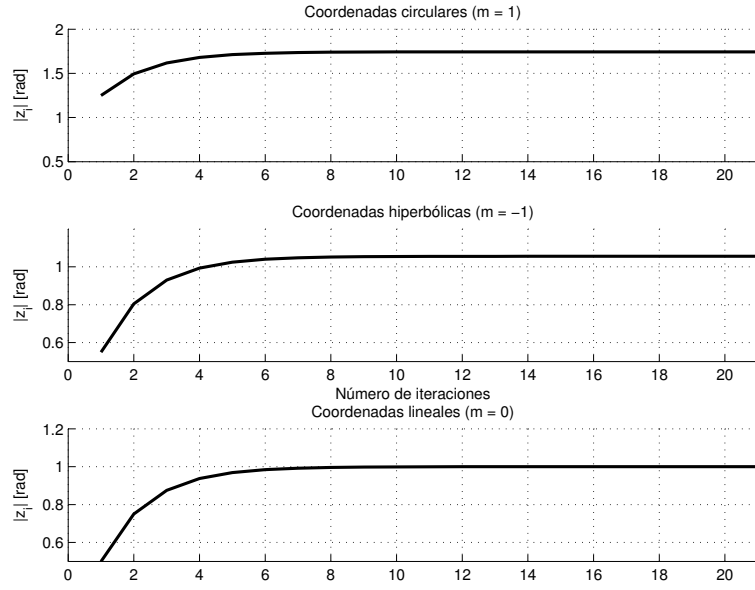
El algoritmo CORDIC converge si en una determinada iteración, la suma de los ángulos de las micro-rotaciones restantes, es suficiente para hacer cero la variable  $z$  en el modo rotación, o la variable  $y$  en el modo vectorización [9].

En la práctica, no siempre es posible hacer cero estas variables, luego el algoritmo convergerá si se acercan a cero con una determinada tolerancia, generalmente el valor de la última micro-rotación.

De forma matemática, podemos expresar la condición de convergencia en el modo *rotación* como:

$$z_{n+1} = \left| z_i - \sum_{j=i}^n \right| \leq \alpha_{n.m} \quad (2.19)$$

Despejando  $z_i$  de la ecuación (2.19), se puede obtener el margen de ángulos de entrada para los que el algoritmo converge. A continuación se muestra el rango de convergencia aproximado para cada sistema



**Figura 2.3:** Rango de convergencia del algoritmo CORDIC, para los diferentes sistemas de coordenadas

de coordenadas:

$$\begin{aligned}
 |z_{i,max}|_{m=0} &\leq \sum_{i=1}^n \alpha_{i,0} + \alpha_{n,0} = \sum_{i=1}^n 2^{-i} \approx 1 \text{ rad} \\
 |z_{i,max}|_{m=1} &\leq \sum_{i=0}^n \alpha_{i,1} + \alpha_{n,1} = \sum_{i=0}^n \tan^{-1} 2^{-i} \approx 1,7433 \text{ rad} \\
 |z_{i,max}|_{m=-1} &\leq \sum_{i=1}^n \alpha_{i,-1} + \alpha_{n,-1} = \sum_{i=1}^n \tanh^{-1} 2^{-i} \approx 1,0554 \text{ rad}
 \end{aligned} \tag{2.20}$$

Representando (2.20) de forma gráfica (figura 2.3), se puede comprobar que, a partir de  $n = 8$  iteraciones, el margen de convergencia se estabiliza en un valor diferente para cada sistema de coordenadas.

En el caso del modo vectorización, el razonamiento es similar. Para comprobar si un determinado vector de entrada convergería, no hay más que sustituir el valor de  $|z_{i,max}|$  en (2.20), por su valor en función de  $x$  e  $y$  (ver 2.6).

Centrándonos en el sistema de coordenadas circulares se deduce que, al ser valor absoluto del ángulo de entrada admisible mayor que  $\frac{\pi}{2}$  rad, cualquier vector de entrada cuya coordenada  $x$  sea positiva convergerá.

#### 2.2.4. Extensión del rango de convergencia

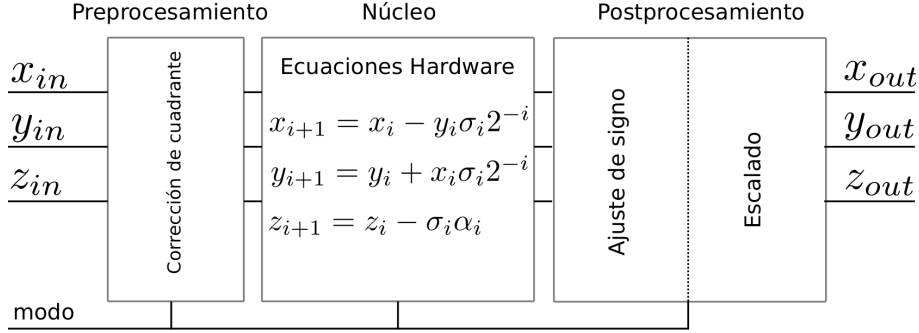
En muchas aplicaciones no es suficiente con un margen de entrada de  $\frac{\pi}{2}$  rad, luego es necesario extender el funcionamiento de CORDIC a los cuatro cuadrantes, es decir, a un rango de  $z_i = [-\pi, \pi]$ . En nuestro caso particular, el algoritmo solo trabajará en coordenadas circulares, por lo que se realizará el desarrollo para este sistema.

Para extender el rango de convergencia, hay que procesar el vector de entrada situándolo dentro del primer o cuarto cuadrante, y teniendo en cuenta esta modificación en el resultado. Dependiendo del

modo de funcionamiento, el proceso varía ligeramente.

En adelante, para evitar confusión entre los valores iniciales del algoritmo y los valores en función de la iteración  $i$ , se denominará a los primeros  $(x_0, y_0, z_0)$ , siendo las variables en función de la iteración  $i$   $(x_i, y_i, z_i)$ .

#### 2.2.4.1. Corrección de cuadrante en el modo rotación



**Figura 2.4:** Diagrama de bloques del algoritmo CORDIC

En la figura 2.4 se muestra un diagrama de bloques general del algoritmo CORDIC. Para realizar la corrección de cuadrante en modo rotación, es necesario modificar el ángulo de entrada  $z_0$  (preprocesamiento) y realizar una corrección del resultado para que sea acorde con el ángulo inicial (postprocesamiento).

En el preprocesamiento, se comprueba el cuadrante en el que se encuentra el ángulo a rotar, trabajando con un ángulo alternativo de  $z'_0 = z_0 \pm \frac{\pi}{2}$ , en el caso de que el vector se encuentre en el segundo o en el tercer cuadrante.

Con esto se consigue que la variable  $z$  entre en la zona de convergencia y, por tanto, pueda ser llevada a cero. Al haber actuado sobre la coordenada  $z$ , el resultado que se obtendrá será la rotación del vector de entrada un ángulo  $z'_0$ , luego es necesario corregirlo.

En la expresión 2.21 podemos observar el valor que tomarán las variables  $x$  e  $y$  tras  $n$  iteraciones, al utilizar el ángulo alternativo  $z'_0$ .

$$\begin{aligned} x_n &= K_1 \cdot \left[ x_0 \cos \left( z_0 - \frac{\pi}{2} \right) - y_0 \sin \left( z_0 - \frac{\pi}{2} \right) \right] \\ y_n &= K_1 \cdot \left[ y_0 \cos \left( z_0 - \frac{\pi}{2} \right) + x_0 \sin \left( z_0 - \frac{\pi}{2} \right) \right] \end{aligned} \quad (2.21)$$

En el caso de que inicialmente el vector se encontrara en el segundo cuadrante, se habrá trabajado con un ángulo de  $z'_0 = z_0 - \frac{\pi}{2}$ . Atendiendo a las identidades trigonométricas:

$$\begin{aligned} \sin \left( \beta - \frac{\pi}{2} \right) &= -\cos \beta \\ \cos \left( \beta - \frac{\pi}{2} \right) &= \sin \beta \end{aligned} \quad (2.22)$$

Se deduce que el cambio a realizar será:

$$\begin{aligned} x'_n &= -y_n \\ y'_n &= x_n \end{aligned} \quad (2.23)$$

Siguiendo un razonamiento similar, cuando el vector de origen se encuentra en el tercer cuadrante, se llega a las expresiones:

$$\begin{aligned} x'_n &= y_n \\ y'_n &= -x_n \end{aligned} \quad (2.24)$$

#### 2.2.4.2. Corrección de cuadrante en el modo vectorización

Cuando el algoritmo va a funcionar en modo vectorización, en coordenadas circulares, se calcula el arcotangente y el módulo de un determinado vector  $\vec{v}_0 = (x_0, y_0)$ .

En el caso de que este vector se encuentre en el segundo o el tercer cuadrante, para que el algoritmo converja es necesario proyectarlo sobre el pimer o cuarto cuadrante, acumulando el ángulo girado en la variable  $z$ .

Para demostrar esto, en primer lugar se presentan las ecuaciones de salida:

$$\begin{aligned} x_n &= K_1 \cdot \sqrt{x_0^2 + y_0^2} \\ z_n &= z_0 + \tan^{-1} \left( \frac{y_0}{x_0} \right) \end{aligned} \quad (2.25)$$

De ellas podemos deducir dos cosas:

- El módulo será el mismo, independientemente de donde se encuentre el vector.
- Añadiendo en  $z_0$  el ángulo rotado para realizar la proyección, obtendremos a la salida la fase del vector inicial.

De forma matemática, si el vector se encuentra inicialmente en el *segundo cuadrante*, los cambios a realizar son:

$$\begin{aligned} x'_0 &= y_0 \\ y'_0 &= -x_0 \\ z'_0 &= z_0 + \frac{\pi}{2} \end{aligned} \quad (2.26)$$

Mientras que, si se encuentra en el *tercer cuadrante*:

$$\begin{aligned} x'_0 &= -y_0 \\ y'_0 &= x_0 \\ z'_0 &= z_0 - \frac{\pi}{2} \end{aligned} \quad (2.27)$$

No siendo necesario realizar ningún cambio en el resultado final.

## 2.3. Algoritmo de Jacobi clásico

Como se ha comentado, la idea fundamental detrás del algoritmo de Jacobi, es diagonalizar una matriz de entrada  $A \in \mathbb{R}^{n \times n}$  iterando sobre ella, eliminando un elemento cada vez. Para ello, en cada iteración se rota un ángulo  $\theta$ , haciendo uso de la matriz de rotación de Givens (2.28).

$$R(\theta) = \begin{bmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & C_\theta & \cdots & S_\theta & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & -S_\theta & \cdots & C_\theta & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{bmatrix} \quad (2.28)$$

Como punto de partida para el desarrollo se toma la definición de autovector:

$$A \cdot V = \lambda I \cdot V \quad (2.29)$$

Donde, si  $A \in \mathbb{R}^{n \times n}$  es simétrica, entonces:

- $\lambda I$  es una matriz diagonal que contiene los  $n$  autovalores de  $A$ .
- $V \in \mathbb{R}^{n \times n}$  es una matriz ortogonal que contiene los  $n$  autovectores correspondientes a los  $n$  autovalores de  $A$ .

Si multiplicamos la expresión 2.29 por  $V^{-1}$  a ambos lados se tiene que:

$$V^{-1}AV = D \quad \text{donde} \quad D \in \mathbb{R}^{n \times n} = \lambda I \quad (2.30)$$

Y elevando (2.30) al cuadrado, se llega a la ecuación (2.31).

$$V^{-1}AVV^{-1}AV = DD \quad (2.31)$$

Pudiéndose comprobar por inducción que:

$$V^{-1}A^aV = D^a \quad (2.32)$$

Además, como la matriz de autovectores  $V$  es ortogonal, su inversa es igual a su transpuesta, es decir  $V^{-1} = V^T$  luego la expresión 2.32, es equivalente a:

$$V^T A^a V = D^a \quad (2.33)$$

Por otro lado, en una matriz cuadrada  $\Sigma \in \mathbb{C}^{n \times n}$ , los valores singulares se definen como la raíz cuadrada de los autovalores de  $\Sigma\Sigma^*$ , siendo  $\Sigma^*$  la transpuesta conjugada de  $\Sigma$  [11]. Particularizando para matrices reales y simétricas, tenemos que, los valores propios de  $A \in \mathbb{R}^{n \times n}$ , se corresponden con los autovalores de  $AA^T$ .

Volviendo a la expresión (2.33) puede verse que, en el caso de  $a = 2$ , los autovalores de  $A$  se corresponden con los valores singulares de  $A^2$ ; teniendo esto en cuenta, podemos deducir una expresión iterativa. Tomando ( $a = 1$ ) en la ecuación (2.33) y añadiendo una variable de iteración ( $k$ ), se tiene que:

$$A_{k+1} = V_k^T \cdot A_k \cdot V_k; \quad k = 0, 1, \dots, h \quad (2.34)$$

Donde  $h$  es el número de iteraciones necesario para considerar  $A$  diagonalizada.

Las matrices  $V_k$  utilizadas durante las rotaciones, son precisamente matrices de rotación de Givens como la mostrada en (2.28), que cumplen la condición de ortogonalidad. Ahora solo queda encontrar un ángulo  $\alpha$  tal que, al realizar la doble multiplicación de la expresión (2.34), se consiga hacer cero un elemento arbitrario.

Este valor puede encontrarse resolviendo el caso bidimensional:

$$D = R(\theta)^T \cdot A \cdot R(\theta) = \begin{bmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{bmatrix}^T \cdot \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \cdot \begin{bmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{bmatrix} \quad (2.35)$$

Si desarrollamos la ecuación (2.35) para determinar el valor que toma el elemento  $d_{12}$  tenemos que:

$$d_{12} = a_{11} \sin \theta \cos \theta - a_{21} \sin^2 \theta + a_{12} \cos^2 \theta - a_{22} \cos \theta \sin \theta \quad (2.36)$$

Puesto que queremos conseguir que los elementos fuera de la diagonal principal sean cero tras las rotaciones, hemos de igualar a cero la expresión 2.36. Por otro lado, para poder despejar  $\theta$ , hay que tener en cuenta que, puesto que las matrices son simétricas,  $a_{12} = a_{21}$ . Además de las identidades trigonométricas:

$$\begin{aligned} 2 \sin \alpha \cos \alpha &= \sin(2\alpha) \\ \cos^2 \alpha - \sin^2 \alpha &= \cos(2\alpha) \end{aligned} \quad (2.37)$$

Tras esto, llegamos a la ecuación 2.38

$$\frac{1}{2} \sin(2\theta)(a_{11} - a_{12}) + a_{12} \cos(2\theta) = 0 \quad (2.38)$$

Dividiendo (2.38) entre  $\cos(2\theta)$  y despejando, se obtiene que el ángulo de rotación debe cumplir la ecuación (2.39).

$$\tan(2\theta) = \frac{2a_{12}}{a_{22} - a_{11}} \quad (2.39)$$

Algunos criterios para seleccionar que elemento es más adecuado eliminar en cada iteración, consisten en la búsqueda del mayor elemento fuera de la diagonal, o iterar sobre todos los elementos de forma sucesiva. Esto implica añadir carga computacional extra al algoritmo. Como se verá a continuación, este problema no existe en la implementación por división en submatrices  $2 \times 2$ .

## 2.4. Algoritmo de Jacobi por división en submatrices $2 \times 2$

Si se estudia el producto  $R^T A R$  puede comprobarse que, los únicos elementos de  $A$  afectados, son los correspondientes a los cuatro elementos de  $R$  que implementan la rotación. Esto implica que es posible realizar rotaciones simultaneas sobre la matriz  $A$ , eligiendo adecuadamente el elemento central de cada una.

En [2] se plantea el problema mediante la división de  $A \in \mathbb{R}^{n \times n}$  en  $\frac{n}{2} \times \frac{n}{2}$  submatrices de dimensiones  $2 \times 2$ , realizando en cada iteración una doble rotación sobre cada una. En la ecuación (2.40), se muestra la matriz  $A$  dividida en  $\frac{n}{2} \times \frac{n}{2}$  submatrices.

$$\left[ \begin{array}{cccc} \begin{pmatrix} a_{11} & & a_{12} \\ & S_{11} & \\ a_{21} & & a_{22} \end{pmatrix} & \begin{pmatrix} a_{13} & & a_{14} \\ & S_{12} & \\ a_{23} & & a_{24} \end{pmatrix} & \cdots & \begin{pmatrix} a_{1,n-1} & & a_{1,n} \\ & S_{1,\frac{n}{2}} & \\ a_{2,n-1} & & a_{2,n} \end{pmatrix} \\ \begin{pmatrix} a_{31} & & a_{32} \\ & S_{21} & \\ a_{41} & & a_{42} \end{pmatrix} & \begin{pmatrix} a_{33} & & a_{34} \\ & S_{22} & \\ a_{43} & & a_{44} \end{pmatrix} & \cdots & \begin{pmatrix} a_{3,n-1} & & a_{3,n} \\ & S_{2,\frac{n}{2}} & \\ a_{4,n-1} & & a_{4,n} \end{pmatrix} \\ \vdots & \vdots & \ddots & \vdots \\ \begin{pmatrix} a_{n-1,1} & & a_{n-1,2} \\ & S_{\frac{n}{2},1} & \\ a_{n,1} & & a_{n,2} \end{pmatrix} & \begin{pmatrix} a_{n-1,3} & & a_{n-1,4} \\ & S_{\frac{n}{2},2} & \\ a_{n,3} & & a_{n,4} \end{pmatrix} & \cdots & \begin{pmatrix} a_{n-1,n-1} & & a_{n-1,n} \\ & S_{\frac{n}{2},\frac{n}{2}} & \\ a_{n,n-1} & & a_{n,n} \end{pmatrix} \end{array} \right] \quad (2.40)$$

Pudiendo reformularse una submatriz cualquiera como:

$$S_{i,j} = \begin{bmatrix} s_{2i-1,2j-1} & s_{2i-1,2j} \\ s_{2i,2j-1} & s_{2i,2j} \end{bmatrix}, i, j = 1, \dots, n/2 \quad (2.41)$$

La realización de  $n/2$  rotaciones en cada iteración, implica el uso de  $n/2$  ángulos de rotación, que se determinan resolviendo el problema bidimensional en las submatrices diagonales, utilizando cada submatriz no diagonal el ángulo correspondiente a su fila y su columna. La ecuación iterativa correspondiente a cada submatriz será por tanto:

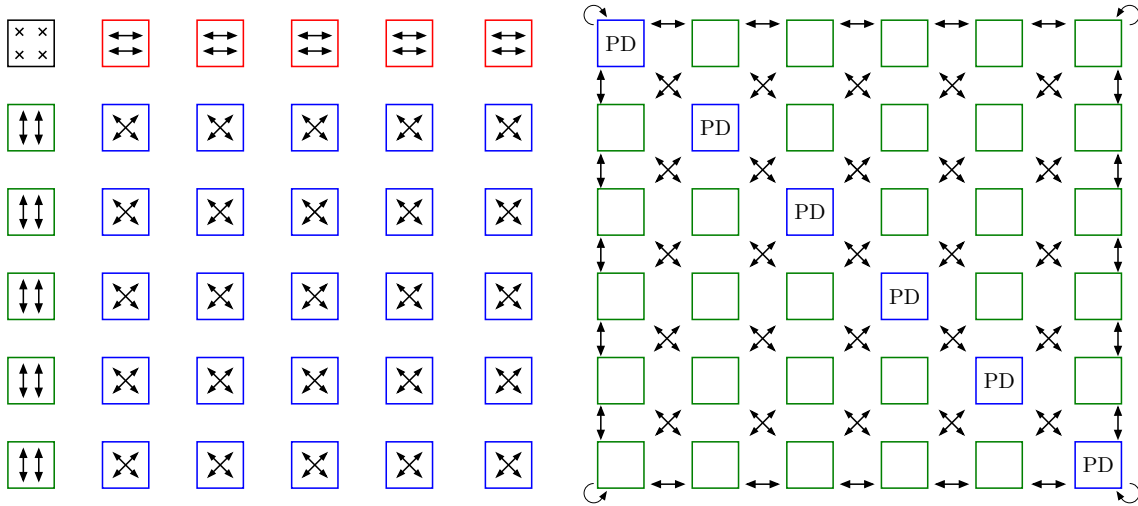
$$S_{ij}^{(k+1)} = R(\alpha_{ii}^{(k)})^T \cdot S_{ij}^{(k)} \cdot R(\alpha_{jj}^{(k)}) \quad (2.42)$$

Obteniéndose los ángulos de rotación  $\alpha_{ii}$  y  $\alpha_{jj}$ , a partir de la expresión (2.39) evaluada en las submatrices diagonales  $S_{ii}$  y  $S_{jj}$ .

En [2], se plantea un diseño que hace uso de módulos hardware interconectados para implementar las operaciones de rotación y el cálculo de ángulos sobre las submatrices diagonales, denominadas procesadores diagonales (PD) y la operación de rotación en las submatrices no diagonales o procesadores no diagonales (PND), realizándose en cada iteración la propagación de los ángulos entre los PD y PND.

Por otro lado, al finalizar el cálculo de la doble rotación, se propone un reordenamiento de los datos entre todas las submatrices para poder eliminar nuevos elementos en la siguiente iteración. Este reordenamiento es necesario puesto que, los elementos que se eliminan en cada iteración, se encuentran en posiciones fijas, siendo necesario mover los elementos no nulos a esas posiciones.

En la figura 2.5 se muestra tanto el reordenamiento interno como el intercambio de datos entre las submatrices adyacentes.



**Figura 2.5:** Reordenamiento interno y externo de los datos tras cada iteración

El cálculo de autovectores se realiza de forma similar, partiendo de una matriz  $V^{(1)} = I_{n \times n}$  (donde  $I$  es la matriz identidad) que es dividida en submatrices  $2 \times 2$ , iterando sobre ella mediante la expresión:

$$V^{(k+1)} = V^{(k)} \cdot R(\alpha_{jj}^{(k)}) \quad (2.43)$$



### 2.4.1. Utilización del algoritmo CORDIC para realizar las dobles rotaciones

Para calcular la doble rotación ( $R^T A R$ ) sobre cada submatriz, en [3] se plantea el uso de algoritmo CORDIC. La ventaja principal es la reducción del número de operaciones, pudiéndose realizar la operación en cuatro ejecuciones del algoritmo.

El objetivo, es abordar la rotación de la matriz  $A$ , como la rotación de una serie de vectores. Para ilustrar esto, se va a desarrollar el producto  $R(\alpha)^T \cdot M$ , donde  $M \in \mathbb{R}^{2 \times 2}$  es una matriz arbitraria.

$$\begin{bmatrix} C_\alpha & -S_\alpha \\ S_\alpha & C_\alpha \end{bmatrix} \cdot \begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix} = \begin{bmatrix} C_\alpha m_{11} - S_\alpha m_{21} & C_\alpha m_{12} - S_\alpha m_{22} \\ S_\alpha m_{11} + C_\alpha m_{21} & S_\alpha m_{12} + C_\alpha m_{22} \end{bmatrix} \quad (2.44)$$

Observando el resultado, puede comprobarse que las columnas de  $R^T M$  se corresponden con las ecuaciones de salida del algoritmo CORDIC (2.2) cuando se realiza la rotación de los vectores,  $\vec{v}_1 = (m_{11}, m_{21})$  y  $\vec{v}_2 = (m_{12}, m_{22})$  un ángulo  $\alpha$ .

De forma análoga, si  $Q = R(\alpha^{(k)})^T \cdot M$ , entonces las filas de  $Q \cdot R$  se corresponden con la rotación de los vectores  $\vec{v}_3 = (q_{11}, q_{12})$  y  $\vec{v}_4 = (q_{21}, q_{22})$ .



# Implementación en Vivado HLS

En este capítulo se presenta la implementación del algoritmo CORDIC y el método de Jacobi para el cálculo de autovalores y autovectores, mediante la herramienta Vivado HLS de Xilinx. La metodología seguida en ambos caso es similar.

Centrándonos en el algoritmo CORDIC, en primer lugar se discuten los detalles del diseño, principalmente la codificación y el ancho de palabra, realizando una primera síntesis en Vivado HLS. A partir del diseño preliminar, se hace un estudio experimental del error cometido, para seleccionar el número de iteraciones adecuado.

Finalmente, se procede a la optimización del sistema, dejándolo listo para su uso en la implementación del algoritmo de Jacobi. Como base para la optimización, se parte de las características de las arquitecturas hardware (serie y paralela) del algoritmo CORDIC.

En cuanto al algoritmo de Jacobi, de igual manera que con el módulo CORDIC, se realiza una implementación inicial a partir de la cual se determina el número adecuado de iteraciones, optimizando posteriormente el diseño.

En el proceso de optimización, se han tomado como base las arquitecturas de Brent [5] y Bravo [12], explorando otras opciones intermedias, gracias a la flexibilidad de la herramienta.

## 3.1. Implementación del algoritmo CORDIC

### 3.1.1. Arquitecturas hardware

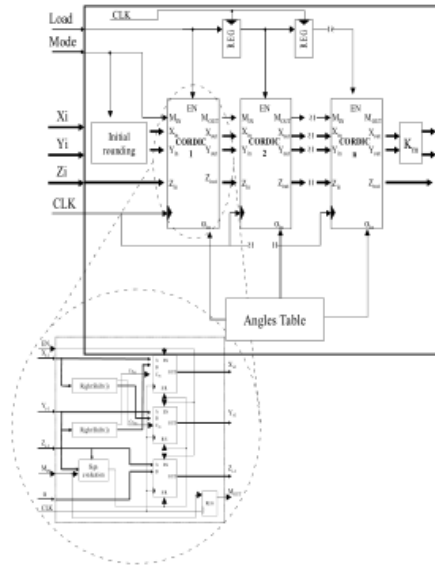
A la hora de abordar la implementación del algoritmo CORDIC en hardware, aparecen dos esquemas fundamentales: la arquitectura paralela y la arquitectura serie.

Si se pretende minimizar el número de recursos consumidos, a costa del tiempo de ejecución, se puede utilizar la arquitectura serie. Por otro lado, si se busca conseguir máximas prestaciones posibles, la arquitectura paralela es la más adecuada.

### 3.1.2. Arquitectura serie

En la arquitectura serie, se implementan las ecuaciones principales del algoritmo, compartiendo el hardware para realizar todas las iteraciones. Es decir, la salida de cada iteración se realimenta a la entrada para de cara a la siguiente.



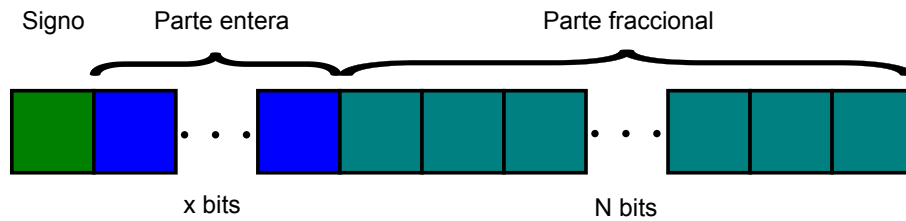


**Figura 3.2:** Arquitectura paralela del algoritmo CORDIC

El ancho de palabra teniendo en cuenta lo anterior, se puede expresar como:

$$WL = 1 + X + N \quad (3.1)$$

En la figura 3.3 se muestra la distribución de bits de forma gráfica.



**Figura 3.3:** Esquema de codificación XQN

La precisión conseguida con las representaciones en coma fija, viene determinada por el número de bits reservado a la parte fraccionaria. En el algoritmo CORDIC, cabe hacer una distinción entre la codificación de los ángulos (variable  $z$ ), y la codificación de las coordenadas del vector (variables  $x$  e  $y$ ), puesto que no hay interacción directa entre ellas:

En el caso de los ángulos, si se trabaja en radianes se presentan dos posibilidades. Por un lado, se pueden utilizar radianes naturales, siendo necesarios al menos dos bits de parte entera y uno de signo, para poder trabajar con valores en el intervalo  $z = [-\pi, \pi]$ . La otra posibilidad, consiste en normalizar los ángulos al intervalo  $z = (-1, 1)$ , siendo necesario únicamente el bit de signo y la parte fraccionaria.

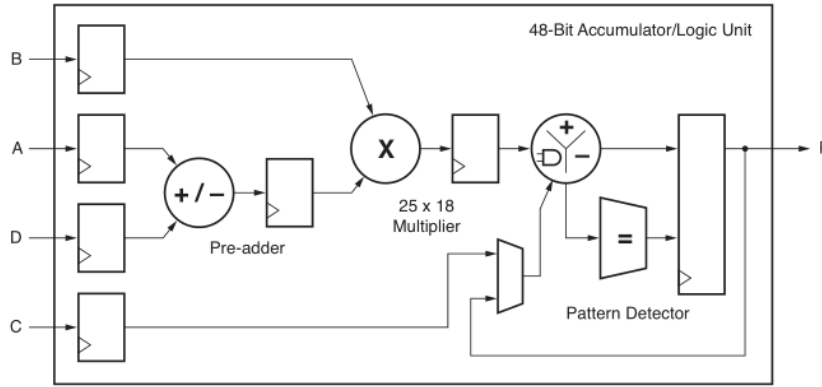
En este trabajo, se ha optado por trabajar con los ángulos sin normalizar, luego la codificación a utilizar será 2QN, para poder manejar valores en el intervalo  $z = [-\pi, \pi]$ :

En el caso de las variables  $x$  e  $y$ , la codificación a utilizar depende de varios factores. En primer lugar, la parte entera debe ser suficientemente grande para evitar desbordamientos en las operaciones internas. En [10] se demuestra que, el tamaño máximo que alcanzarán las variables  $x$  e  $y$ , depende de su valor inicial y la constante  $K_m$ .

Particularizando para el sistema de coordenadas circulares, si se trabaja con valores de  $x$  e  $y$  en un intervalo de  $\pm 1$ , el valor máximo alcanzado será de  $K_1\sqrt{2}$ , luego serán necesarios 2 bits de parte entera y uno de signo para codificarlos.

Puesto que el rango de  $\pm 1$  se considera suficiente para la aplicación, se tomará como punto de partida, siendo el formato de codificación utilizado 2QN. En el caso de que fuera necesario manejar valores de entrada mayores, la solución pasa por normalizar los datos, preferiblemente mediante un factor de  $2^{-k}$ , para evitar el uso de multiplicadores.

Concretado el número de datos de parte entera que se requiere para codificar las diferentes variables, solo queda elegir el ancho de palabra  $WL$ .



**Figura 3.4:** Esquemático del módulo DSP48E1 de la serie 7 de Xilinx

A la hora de elegir el ancho de palabra para implementar CORDIC en una FPGA, se ha decidido atender al número de bits de entrada de los multiplicadores hardware internos. En las FPGA de la serie 7 de Xilinx, todos los multiplicadores se encuentran dentro de módulos DSP48E, cuya estructura interna puede verse en la figura 3.4.

Atendiendo a lo expuesto anteriormente, si se pretende minimizar el número de multiplicadores hardware consumidos, el tamaño de datos que se maneje deberá ser de 18 bits, acorde con las entradas del multiplicador, luego:

$$WL = 18\text{bit} \quad (3.2)$$

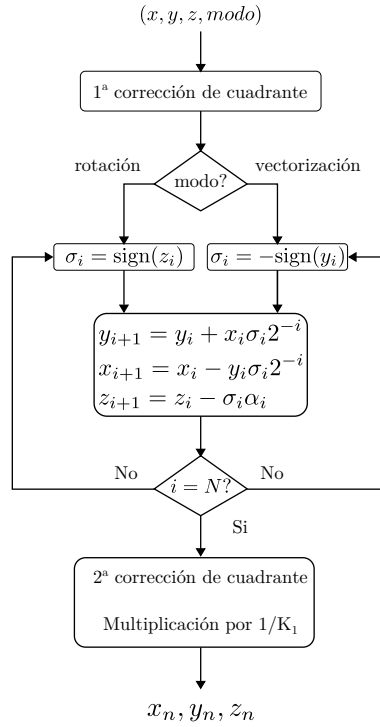
### 3.1.5. Implementación algorítmica

Como ya se ha comentado, en Vivado HLS se utilizan lenguajes de programación de alto nivel, concretamente en C/C++. En este trabajo se ha hecho uso únicamente del lenguaje C.

Una vez determinados los parámetros básicos del diseño, el primer paso de la implementación es codificar y validar una función en C, que capture la funcionalidad del algoritmo. En el apéndice A se comentan varios ejemplos del funcionamiento de la síntesis de alto nivel en Vivado HLS y la estructura jerárquica de los diseños.

En la figura 3.5, se ha representado un diagrama funcional del algoritmo CORDIC, a partir de lo expuesto en la sección 2.2. Tomando esto como base, el primer paso es determinar los argumentos de la función principal (*top function*), a la que se ha denominado `cordic_scp`.

Estos parámetros, se corresponderán con los puertos de entrada/salida del sistema hardware que se generará posteriormente. Según el diagrama, necesitamos tres variables de entrada y tres salidas, que se representan como pasos por referencia. Además de una variable booleana de control, que determinará el modo (rotación o vectorización) en el que se opera.



**Figura 3.5:** Diagrama funcional del algoritmo CORDIC para N iteraciones

A continuación se muestra el prototipo de la función:

```

void cordic_scp(cordic_t xi, cordic_t yi, cordic_t zi,
               cordic_t *xo, cordic_t *yo, cordic_t *zo,
               uint1 mode);
  
```

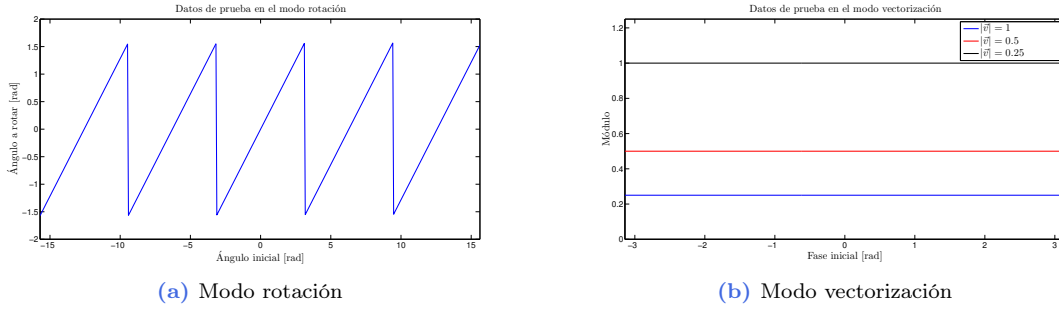
Donde `cordic_t` es un tipo de datos de 18 bits, acorde con lo expuesto en la sección anterior.

Como comentario adicional, los ángulos correspondientes a las micro-rotaciones, se implementan mediante un array que, dependiendo de las directivas de optimización aplicadas, se comportará como una memoria ROM de uno o dos puertos, o un banco de registros.

En primer lugar, se implementa la corrección de cuadrante si es necesario. Tras esto, el comportamiento iterativo se describe mediante un bucle, que evalúa las *ecuaciones hardware* (2.12) para  $i = 0, \dots, \text{MAX\_ITERATIONS}$ . Finalmente, se realiza la segunda corrección de cuadrante y el escalado del vector de salida, mediante la multiplicación directa por  $1/K_1$ .

De cara al resto del desarrollo, conviene comentar la forma que tiene Vivado HLS de evaluar los bucles:

- Cuando se diseña hardware utilizando lenguajes de descripción RTL, como VHDL, los bucles representan una estructura repetitiva, como por ejemplo las etapas de un filtro FIR o las etapas de la arquitectura paralela del algoritmo CORDIC.
- En un lenguaje de programación de propósito general, como C, los bucles representan una secuencia de operaciones que se ejecutan de forma secuencial, no habiendo paralelismo ni entre iteraciones ni en las operaciones internas.
- En Vivado HLS, cuando se realiza la conversión a hardware de una función C que contiene un bucle, pueden presentarse diferentes situaciones, en función de las directivas dadas por el usuario:
  - Puede haber paralelismo entre bucles.



**Figura 3.6:** Datos de entrada para validar el algoritmo CORDIC

- Las operaciones dentro de un bucle se pueden ejecutar de forma concurrente si las dependencias de datos lo permiten.
- Se pueden ejecutar las diferentes iteraciones de forma concurrente.

Controlar este comportamiento cuando se optimiza el diseño, es vital para obtener una implementación eficiente, puesto que ambos algoritmos se fundamentan en el uso de bucles para describir un comportamiento iterativo.

Una vez realizada la implementación software, es necesario someterla a una cantidad suficiente de datos de entrada que permita evaluar todas las situaciones posibles, para asegurar el correcto funcionamiento a nivel algorítmico del sistema.

De forma similar a lo que ocurre con los lenguajes de descripción hardware, donde para realizar la validación de un bloque funcional se recurre a un diseño de mayor jerarquía (*test bench*), cuando se trabaja mediante síntesis de alto nivel utilizando C, la *top function* (`cordic_scp`) se somete a prueba en la función `main()`, no existiendo limitaciones en el lenguaje.

Para generar los datos de prueba y los resultados esperados, se ha utilizado MATLAB. Por un lado, se ha implementado y validado una función equivalente, que puede operar en coma fija o coma flotante. Por otro lado, se han creado sendos ficheros con datos de prueba para los modos de vectorización y rotación.

- En el caso del modo *rotación*, se han generado una serie de vectores con una fase  $z$  inicial comprendida entre  $\pm\pi$  rad, realizando rotaciones en un intervalo de  $\pm\frac{\pi}{2}$  rad, conforme a la figura 3.6a.
- En modo *vectorización*, se ha sometido a prueba utilizando vectores de módulo fijo, partiendo de diferentes ángulos iniciales en el intervalo  $\pm\pi$  (figura 3.6b).

Finalmente, comprobada la validez algorítmica del diseño, se realiza una síntesis en Vivado HLS, sin llevar a cabo ninguna optimización. Finalizada la síntesis, los resultados obtenidos son los mostrados en la tabla 3.1. Tras esto, se está en condiciones de realizar una co-simulación, en la que se comprueba que el diseño RTL conserva la funcionalidad del algoritmo.

El dispositivo elegido para la generación del diseño RTL, es la FPGA que se encuentra en la plataforma de desarrollo Nexys 4 de Diligent [13], por tener un coste moderado y pertenecer a la gama media de la nueva serie 7. Concretamente, es el modelo xc7a100csg324-1, de la familia Artix 7. Por otro lado, se ha fijado como restricción un periodo de reloj de  $T_{CLK} = 10\text{ns}$ .

La co-simulación se lleva a cabo reutilizando el mismo *test bench* que en la validación algorítmica, que es convertido automáticamente a VHDL, e introducido en el simulador XSIM junto con el código VHDL generado.

Este proceso, es el equivalente a la simulación funcional (*behavioral*), llevada a cabo habitualmente



	BRAM_18K	DSP48E	FF	LUT	$T_L [T_{clk}]$	$\Pi[T_{clk}]$
Sin optimizar	1	2	165	1036	37	38

**Tabla 3.1:** Resultados de la síntesis del módulo CORDIC en Vivado HLS, sin aplicar ninguna optimización

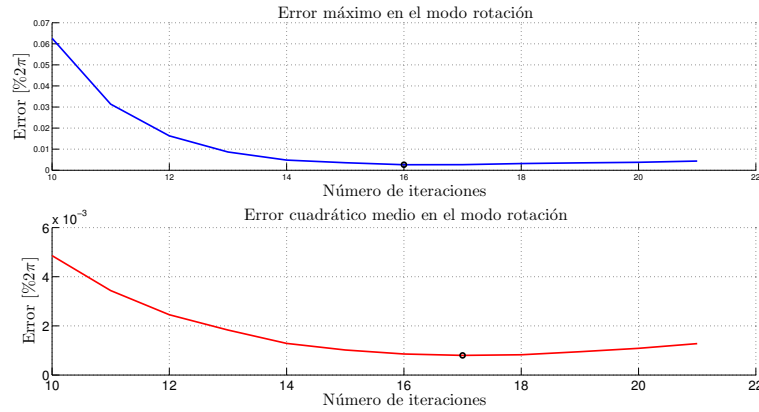
en el flujo de desarrollo RTL, con la salvedad de que se tienen en cuenta tolerancias en el periodo de reloj, para el dispositivo seleccionado.

### 3.1.6. Determinación del número óptimo de iteraciones

Una vez implementado y validado el diseño, estamos en condiciones de determinar el número óptimo de iteraciones a realizar por el algoritmo CORDIC para minimizar el error. En las implementaciones hardware a bajo nivel, un criterio en el caso de la arquitectura paralela es seleccionar el número de etapas del algoritmo acorde con el ancho de palabra  $WL$  del diseño, realizando  $n = WL$  iteraciones en el modo rotación y  $n - 1$  en el modo vectorización.

Puesto que en Vivado HLS no existe este control de bajo nivel, se ha recurrido a realizar simulaciones del hardware variando diferentes parámetros, para justificar el número óptimo de iteraciones. Para el cálculo del error, se han comparado los resultados obtenidos en MATLAB trabajando en coma flotante de doble precisión, con los resultados en coma fija obtenidos en Vivado HLS.

Partiendo de los datos de prueba utilizados en la validación, se han simulado ambos modos de funcionamiento, variando el número de iteraciones y los parámetros asociados a el. En primer lugar, en la figura 3.7 se muestran los resultados obtenidos en el modo rotación.



**Figura 3.7:** Error cometido en el modo rotación en función del número de iteraciones

Calculándose el error porcentual máximo conforme a la expresión:

$$e_{MAX} = \max(e\%)$$

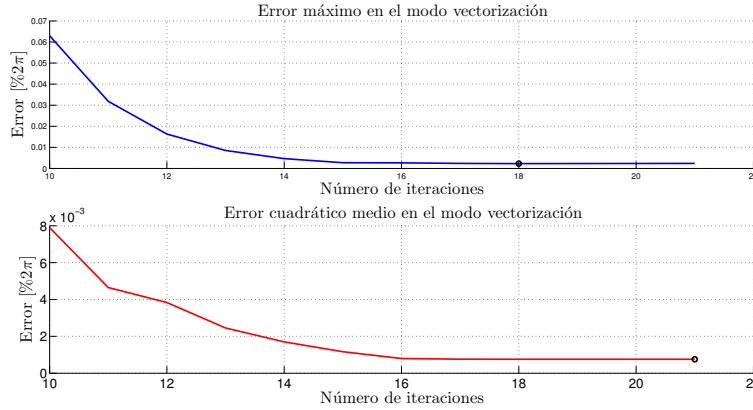
$$e\% = \frac{|V_{HLS}| - |V_{MATLAB}|}{2\pi} \cdot 100 \quad (3.3)$$

Siendo el error cuadrático medio:

$$e_{SQRT} = \frac{1}{k^2} \sqrt{\sum_{i=1}^k e_{MAX}^2} \quad (3.4)$$

Donde  $k$  se corresponde con el número total de muestras. En la expresión (3.3),  $V_{HLS}$  representa el resultado obtenido en la co-simulación del sistema implementado VHDL y  $V_{MATLAB}$ , el resultado obtenido en la simulación en coma flotante mediante MATLAB.

En la gráfica puede observarse un mínimo en el error máximo ( $e_{MAX}$ ) para 16 iteraciones, determinándose que éste es el número óptimo de iteraciones para el modo rotación si lo que se busca es minimizar el error.



**Figura 3.8:** Error cometido en el modo vectorización en función del número de iteraciones

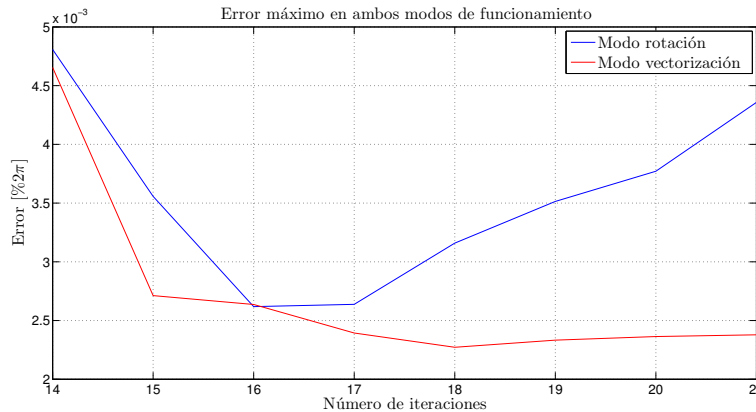
Pasando al modo vectorización, se han obtenido los resultados mostrados en la figura 3.8, procediendo de forma similar. En este caso el mínimo error máximo ( $e_{MAX}$ ) se encuentra a 18 iteraciones.

Una vez espuestos los datos, se pasa a razonar el número adecuado de iteraciones. En la figura 3.9 puede verse el error máximo en ambos modos de funcionamiento de forma simultánea.

Dado que el error en modo rotación para 17 iteraciones es similar al error mínimo, obtenido con 16 iteraciones, y en el modo vectorización el error para 17 iteraciones es también similar al mínimo, obtenido en 18 iteraciones, se ha decidido utilizar un número de iteraciones igual en ambos modos, de 17 iteraciones.

En resumen, el módulo CORDIC finalmente tiene las siguientes características:

- Ancho de palabra ( $WL$ ) de 18 bits.



**Figura 3.9:** Error máximo en ambos modos de funcionamiento en función del número de iteraciones

- $n = 17$  iteraciones en ambos modos de funcionamiento.

### 3.1.7. Optimización del diseño

De los resultados obtenidos a partir de la síntesis del diseño sin optimizar, se deduce rápidamente que la implementación realizada por Vivado HLS es conservadora, minimizando el consumo de recursos. Además, el nivel de paralelismo alcanzado es mínimo.

En esta sección, se realizan sucesivas optimizaciones mediante la inclusión de directivas, con el objetivo de explorar diferentes posibilidades de optimización del diseño. Para ello, se han tenido presentes las arquitecturas hardware presentadas en la sección 3.1.1.

Antes de continuar, aunque en el apéndice A se tratan en detalle, conviene comentar las principales directivas de optimización utilizadas que hay disponibles para mejorar las prestaciones de los diseños.

- La directiva `set_directive_pipeline` (`pipeline`) permite, en un determinado ámbito (una función, un bucle o un bloque de código), maximizar el paralelismo entre las operaciones. Todos los bucles que se encuentren dentro del área de aplicación serán deshechos, es decir, sus iteraciones se implementarán con recursos independientes.
- La directiva `set_directive_unroll` (`unroll`), deshace de forma explícita un bucle, pero, en el caso de los bucles anidados, si se aplica al más exterior los demás seguirán compartiendo el hardware entre iteraciones.
- Las directivas `set_directive_array_partition` y `set_directive_array_reshape` toman un array, por defecto implementados como memorias (ya sea mediante LUT o mediante BRAM), y lo implementan en múltiples memorias (o una con mayor ancho de palabra) o como un banco de registros, para mejorar el ancho de banda. La implementación como banco de registros puede ocurrir de forma automática cuando se aplican las directivas `unroll` o `pipeline` a una zona de código, como se verá más adelante.

Regresando al diagrama de la figura 3.5, podemos determinar que la mayor carga computacional del algoritmo se encuentra en el bucle que implementa las *ecuaciones hardware*, por lo que es el primer lugar donde deberá actuarse.

Si se piensa en la *arquitectura serie*, ésta se fundamenta en un único módulo hardware, que realiza los cálculos asociados a las ecuaciones, realimentando a la entrada los resultados en cada iteración.

Partiendo de este principio, se realiza una implementación añadiendo la directiva `pipeline` al bucle principal, con la que se asegurará alcanzar el máximo paralelismo posible en cada iteración, manteniendo un consumo comedido de recursos. En la tabla 3.2 se muestran los resultados junto con los obtenidos anteriormente para la opción sin optimizar.

	Sin optimizar	<i>Pipeline</i> bucle
BRAM_18K	1	1
DSP48E	2	2
FF	165	129
LUT	1036	1041
$T_L [T_{CLK}]$	37	21
$II [T_{CLK}]$	38	22

**Tabla 3.2:** Resultados de la síntesis en Vivado HLS sin optimizar y con *pipeline* en el bucle principal.

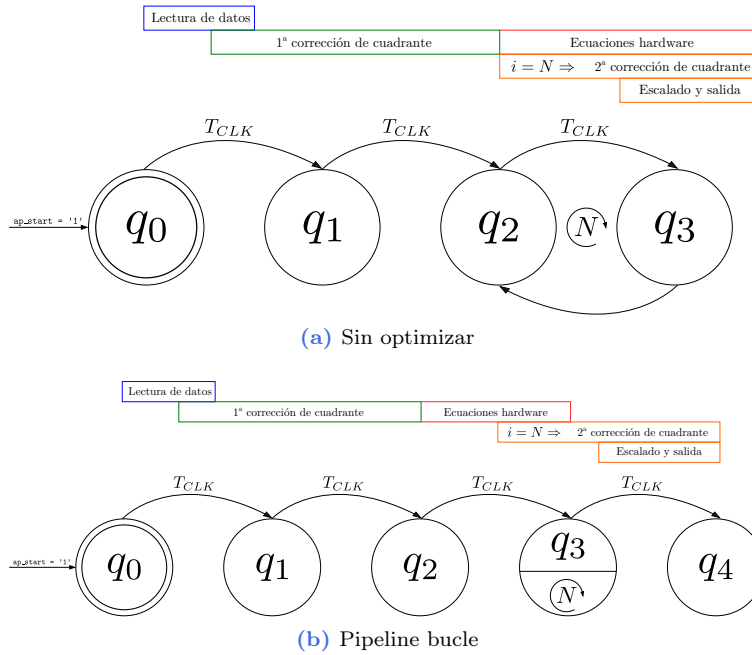
En primer lugar, si se atiende al consumo de recursos, se puede comprobar como la solución (*Pipeline bucle*) presenta unas mejores características, empleando un número similar de elementos discretos.

Esto es debido al paralelismo conseguido mediante la directiva `pipeline`, que hace un mejor uso de los recursos que implementan las operaciones internas del bucle.

El análisis del tiempo de ejecución del sistema generado, puede llevarse a cabo utilizando la perspectiva de análisis de Vivado HLS (ver A.2). De forma general, cuando existen bucles en un diseño, este se implementa como una máquina de estados finita FSM<sup>1</sup>(figura 3.10).

Para evaluar el rendimiento de un diseño, hay que tener en cuenta dos conceptos utilizados en Vivado HLS y, en general, en el diseño digital:

- El *periodo de latencia* ( $T_L$ ), expresado en ciclos de reloj ( $T_{CLK}$ ), se corresponde con el tiempo que transcurre entre la puesta en marcha del sistema y la obtención del primer resultado válido.
- El *Intervalo de Iniciación* (II) representa el número de ciclos de reloj ( $T_{CLK}$ ) que, una vez transcurrido el periodo de latencia, son necesarios para obtener nuevos resultados.



**Figura 3.10:** Maquinas de estados generadas en el proceso de temporización del algoritmo CORDIC

A partir de las máquinas de estados, pueden derivarse sendas expresiones para el periodo de latencia ( $T_L$ ), en función del numero de iteraciones ( $n$ ). En el caso sin optimizar se llega a la ecuación:

$$T_L = 2T_{CLK} + 2nT_{CLK} + T_{CLK} \quad (3.5)$$

Mientras que, cuando se sintetiza el diseño optimizado con *pipeline* en el bucle principal, la latencia puede calcularse como:

$$T_L = 2T_{CLK} + T_{L,loop} + (n - 1)T_{CLK} + T_{CLK} \quad (3.6)$$

Donde  $T_{L,loop} = 2T_{CLK}$  representa la latencia inicial del hardware que implementa el bucle, procesándose un dato por ciclo de reloj, transcurrido un tiempo de  $T_{L,loop}$ .

Si se observan los resultados obtenidos puede verse que, aunque se ha conseguido paralelizar en cierta medida las operaciones del interior del bucle, solo es posible obtener un resultado cada 22 ciclos de reloj. Para conseguir acelerar la ejecución, es necesario deshacer el bucle (*unroll*), implementando cada iteración como un sistema hardware independiente.

<sup>1</sup>Finite State Machine

Si se estudian las dependencias de datos entre dos iteraciones, es obvio que el resultado de la iteración  $i$ , es necesario para comenzar la iteración  $i + 1$  por lo que, aunque se implementen las  $n$  iteraciones como  $n$  etapas, en el mejor de los casos se llegará a un sistema que proporcione un resultado por ciclo de reloj, tras una determinada latencia inicial.

Esta situación es similar a la de la arquitectura paralela, presentada en la sección 3.1.1. Para deshacer el bucle que implementan las ecuaciones hardware, aparecen dos opciones: aplicar directamente una directiva de *unroll* sobre el mismo, o aplicar la directiva *pipeline* a la función `cordic_scp`, puesto que esta deshará todos los bucles interiores.

Tras estudiar los resultados al aplicar *unroll* al bucle principal pudo comprobarse que, a pesar de repetirse  $n$  veces el conjunto de recursos que implementan las ecuaciones hardware, la temporización respecto a la alternativa anterior no mejoró. Analizando el diseño, se llega a la conclusión de que es necesario aplicar la directiva *pipeline* sobre la función principal, para que el sintetizador implemente el sistema de la forma más concurrente posible.

En la tabla 3.3, se compara el consumo de recursos al aplicar *unroll* al bucle principal o *pipeline* a la función completa. Como se puede observar, la latencia ( $T_L$ ) de ambas soluciones es igual, siendo el intervalo de iniciación en la opción totalmente segmentada de un solo ciclo de reloj, es decir, se genera un resultado por ciclo de reloj tras la latencia inicial.

	<i>Unroll</i> bucle	<i>pipeline</i>
BRAM_18K	0	0
DSP48E	2	2
FF	1844	1932
LUT	5082	5086
$T_L [T_{clk}]$	19	19
II $[T_{clk}]$	20	1

**Tabla 3.3:** Resultados de la síntesis en Vivado HLS de CORDIC deshaciendo el bucle principal

Si se atiende al consumo de recursos, ambas alternativas requieren aproximadamente el mismo número de elementos discretos, siendo las prestaciones temporales de la segunda muy superiores, gracias a la posibilidad de recibir nuevos datos continuamente.

### 3.1.8. Interfaz

Por último, antes de proceder con la implementación del algoritmo de Jacobi, conviene comentar la interfaz del módulo CORDIC diseñado. Dependiendo de como se utilice, cabe distinguir entre la interfaz software y la interfaz hardware.

La interfaz software se corresponde con el prototipo de la función principal `cordic_scp` ya comentado:

```
void cordic_scp(cordic_t xi, cordic_t yi, cordic_t zi,
               cordic_t *xo, cordic_t *yo, cordic_t *zo,
               uint1 mode);
```

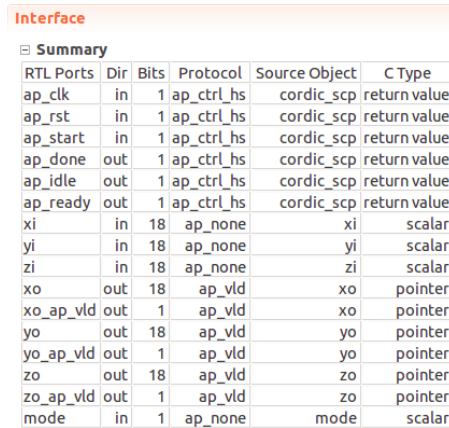
Donde:

- `xi`, `yi` y `zi` representan las variables  $x, y, z$  de entrada.
- `*xo`, `*yo` y `*zo` representan las variables de salida. En este caso, es necesario el uso de punteros para que el sintetizador de Vivado HLS interprete los argumentos como salidas en la interfaz RTL.
- `mode` es una variable de control para seleccionar el modo rotación (`mode = 1`) o el modo vectorización (`mode = 0`).

De cara al resto del diseño, bastará con utilizarla como una función más.

Si se requiere utilizar el módulo hardware en otra herramienta, puede ser exportado como una serie de ficheros VHDL, o directamente como un *core* para su uso en XSG, Vivado o EDK.

Si no se especifica lo contrario, la interfaz que implementará Vivado HLS a partir del prototipo de la función, se compone de un conjunto de puertos de entrada/salida correspondientes a los argumentos, junto con una serie de señales de control. En la figura 3.11, se muestran los puertos del diseño RTL generado por Vivado HLS.



The image shows a screenshot of the 'Interface' tab in Vivado HLS, specifically the 'Summary' section. It displays a table of RTL ports for the CORDIC module. The table has six columns: RTL Ports, Dir, Bits, Protocol, Source Object, and C Type. The ports are categorized into control signals (ap\_\*) and data signals (xi, yi, zi, xo, yo, zo, mode).

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	cordic_scp	return value
ap_rst	in	1	ap_ctrl_hs	cordic_scp	return value
ap_start	in	1	ap_ctrl_hs	cordic_scp	return value
ap_done	out	1	ap_ctrl_hs	cordic_scp	return value
ap_idle	out	1	ap_ctrl_hs	cordic_scp	return value
ap_ready	out	1	ap_ctrl_hs	cordic_scp	return value
xi	in	18	ap_none	xi	scalar
yi	in	18	ap_none	yi	scalar
zi	in	18	ap_none	zi	scalar
xo	out	18	ap_vld	xo	pointer
xo_ap_vld	out	1	ap_vld	xo	pointer
yo	out	18	ap_vld	yo	pointer
yo_ap_vld	out	1	ap_vld	yo	pointer
zo	out	18	ap_vld	zo	pointer
zo_ap_vld	out	1	ap_vld	zo	pointer
mode	in	1	ap_none	mode	scalar

Figura 3.11: Interfaz RTL del módulo CORDIC diseñado

Además de los puertos extraídos directamente del prototipo de la función, aparecen señales de dato valido ([salida]\_ap\_vld), que indican cuando el resultado en la salida correspondiente está listo para ser capturado.

Por otro lado, encontramos una señal de habilitación **ap\_start** y señales que indican cuando el bloque está listo para recibir nuevos datos (**ap\_ready**), cuando está detenida la ejecución (**ap\_idle**), y cuando ha terminado de procesarse un set de datos de entrada (**ap\_done**).

Esta interfaz puede moldearse mediante directivas de optimización, para adaptarse a las necesidades de la aplicación. Por ejemplo, es posible agrupar los parámetros de la función en un bus AXI4 para conectar el sistema como acelerador hardware de un microprocesador.

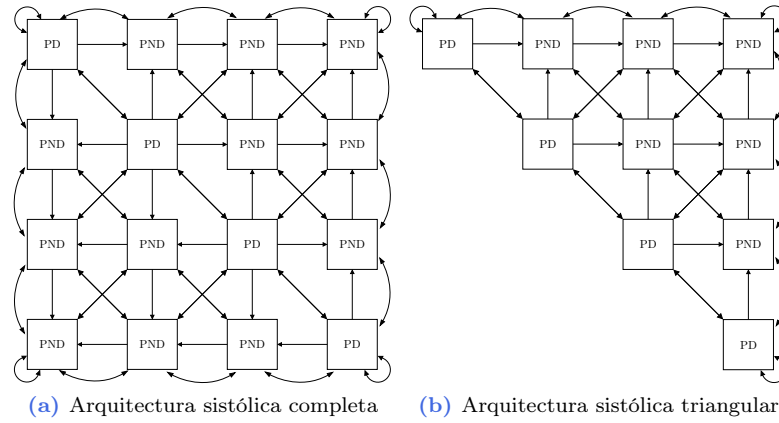
## 3.2. Implementación del algoritmo de Jacobi

A lo largo de esta sección, se discute la implementación del algoritmo de Jacobi por división en submatrices de dimensiones  $2 \times 2$ . Como se comentó anteriormente, hay dos trabajos previos que se utilizan de punto de partida, la propuesta de Brent y Luk en [2], y la propuesta de Bravo en [6].

De forma similar al desarrollo del algoritmo CORDIC, en primer lugar se presentan brevemente las arquitecturas hardware del algoritmo de Jacobi, comentando después los detalles de la implementación, la optimización del diseño y la interfaz.

### 3.2.1. Arquitecturas Hardware

Como se vio en la sección anterior, cuando se implementan algoritmos de tipo computacional sobre hardware a medida aparecen, a grandes rasgos, dos arquitecturas genéricas: paralela y serie.



**Figura 3.12:** Arquitectura sistólica para el cálculo de autovalores y autovectores en matrices simétricas de forma concurrente.

La propuesta de Brent y Luk se encuentra en este primer grupo, implementándose el sistema como una serie de módulos hardware independientes conectados entre sí. La principal ventaja de esta arquitectura, es su reducido tiempo de ejecución. Como contra, el número de recursos consumidos es muy elevado, creciendo con el orden  $n$  de las matrices a razón de  $O(n^2)$ .

Para reducir el consumo de recursos, en el caso de trabajar con matrices simétricas es posible operar solo sobre la mitad triangular superior para realizar el cálculo de autovalores, reduciendo así el número de procesadores no Diagonales (PND). En el caso del cálculo de autovectores, es necesario utilizar el esquema completo. En la figura 3.12 se muestran ambas arquitecturas.

Pasando a las arquitecturas tipo serie, el objetivo es realizar las mismas operaciones que en su equivalente paralelo, reduciendo el número de elementos de cálculo. En el caso del algoritmo de Jacobi, esto significa realizar todas las operaciones haciendo uso de uno o dos procesadores, dependiendo del planteamiento.

Por un lado, se puede implementar un sistema que implemente un PD y un PND, realizando las operaciones correspondientes a la arquitectura paralela de forma secuencial [14]. La otra posibilidad, presentada en [6], es utilizar un procesador que haga las veces de PD y PND, tratando de minimizar el tiempo de ejecución mediante el estudio de las dependencias de datos.

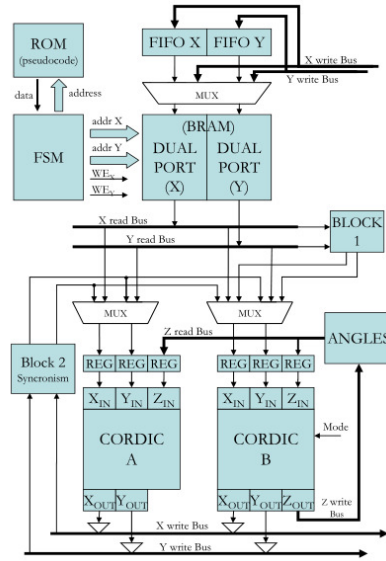
En la figura 3.13 se muestra el sistema serie desarrollado en [6], en el que se utilizaba un núcleo hardware que hacía las veces de procesador diagonal y no diagonal.

A grandes rasgos, en cada iteración se comparten los recursos tanto para determinar los  $n/2$  ángulos de rotación, como para el cálculo de las rotaciones de autovalores y autovectores, utilizándose dos módulos CORDIC para realizar todas las operaciones junto con una serie de bloques de control.

De forma resumida, al comenzar la ejecución se leía la matriz  $A$  de una memoria de doble puerto, realizando el cálculo de los ángulos en uno de los módulos CORDIC. Debido a las dependencias de datos, tras la obtención del primer ángulo de rotación no era posible comenzar el cálculo de autovalores, por lo que se comenzaba con el cálculo de autovectores.

Una vez se disponía de los datos necesarios, se detenía el cálculo de autovectores y se comenzaba a realizar las rotaciones de autovalores. Finalizado el cálculo de los autovectores, se pasaba a realizar un reordenamiento y reducción de los autovalores (tomando solo los de mayor peso), mientras se finalizaba el cálculo de autovectores.

Como se justificó en el capítulo anterior, la multiplicación por la matriz de Givens  $R(\alpha)$ , se corres-



**Figura 3.13:** Arquitectura serie para el cálculo de autovalores y autovectores

ponde con la rotación de dos vectores. En este diseño se utilizaba una memoria DP<sup>2</sup> que permite la lectura/escritura de dos datos por ciclo de reloj mientras que, los módulos CORDIC, necesitaban dos datos en cada ejecución (además de los ángulos ya calculados).

Para abordar el problema, se recurrió al uso de una segunda señal de reloj para atacar a la memoria DP, con el doble de frecuencia que la señal de reloj principal de la FPGA, pudiéndose así leer los cuatro datos en un ciclo de reloj (de cara al resto del sistema).

### 3.2.2. Implementación algorítmica

Puesto que en Vivado HLS el diseño se describe de forma algorítmica, se ha realizado una implementación genérica, siguiendo el método por división en matrices de dimensiones  $2 \times 2$ .

Una vez validada esta implementación, puede ser optimizada para tomar características similares a las arquitecturas hardware presentadas en la sección anterior. Al igual que con el algoritmo CORDIC, se ha realizado un diagrama de bloques funcional (figura 3.14) con la secuencia de operaciones a realizar, como punto de partida para la codificación.

En primer lugar, se recibe una matriz  $A_{n \times n}$  sobre la que se ha de realizar el cálculo de autovalores y autovectores. Esta matriz es almacenada en un array bidimensional  $S$ , con el objetivo de reducir al mínimo los accesos a memoria externa y facilitar la optimización. Por otro lado, se carga la matriz  $V$  de partida para el cálculo de autovectores con una matriz identidad  $I_{n \times n}$ .

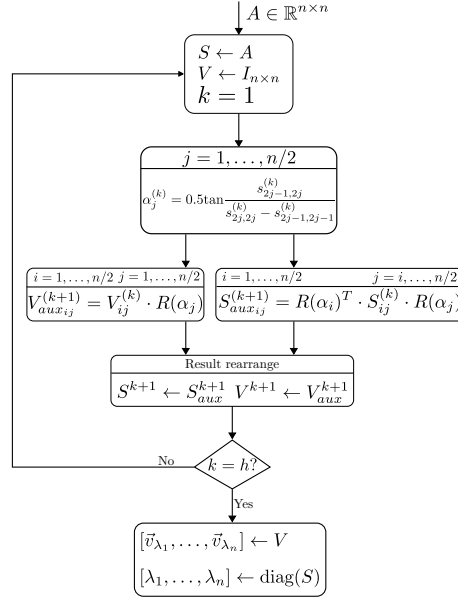
A partir de la matriz  $S$ , se calculan los  $n/2$  ángulos de rotación, correspondientes a las submatrices diagonales  $S_{jj}$  mediante la expresión 2.39, utilizando el algoritmo CORDIC en modo vectorización.

Una vez obtenidos los ángulos de rotación, se itera sobre las  $\frac{n}{2} \times \frac{n}{2}$  submatrices de  $S$  y  $V$  mediante las ecuaciones 2.42 y 2.43. Para realizar los cálculos, como se expuso en la sección 2.4, se hace uso del algoritmo CORDIC.

Al finalizar cada iteración, tanto en el caso de los autovectores como en el caso de los autovalores, el resultado se almacena siguiendo el criterio de ordenamiento ya mostrado en la figura 2.5. Para implementar el reordenamiento, se ha hecho un estudio de todos los pasos del mismo, llegándose a la

<sup>2</sup>Dual Port





**Figura 3.14:** Diagrama de bloques funcional del algoritmo de Jacobi por división en matrices de  $2 \times 2$

conclusión de que se traduce en un intercambio de filas y columnas, tal y como se muestra en la figura 3.15.

Tras la implementación inicial, se valida el algoritmo utilizando una serie de matrices de prueba que permitan asegurar su correcta funcionalidad. Para ello, al igual que con el algoritmo CORDIC, se ha codificado en MATLAB una función similar a la que implementa el diseño en C.

Por otro lado, para disponer de datos de test válidos, se ha hecho uso de la función `eig()` de MATLAB, que calcula tanto los autovectores como los autovalores de una matriz dada.

Una vez validado el diseño, se realiza una primera síntesis en Vivado HLS, seguida de una co-simulación (mediante XSim), utilizando los mismos datos que para la verificación software, para comprobar que el sintetizador ha capturado correctamente la funcionalidad descrita. En la tabla 3.4 se muestran los resultados del sistema sin optimizar.

BRAM	DSP48E	FF	LUT	$T_L [T_{CLK}]$
6	4	2291	8058	16246

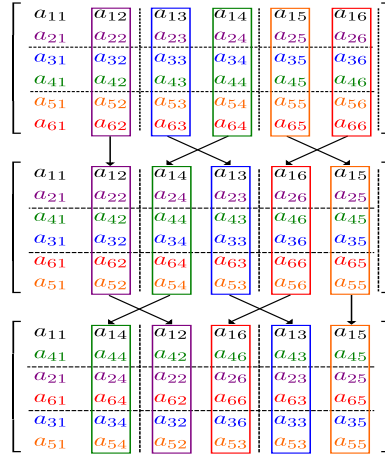
**Tabla 3.4:** Resultados de la síntesis en Vivado HLS del algoritmo de Jacobi, sin realizar ninguna optimización

Tras esto, ya se está en condiciones de analizar el error, para determinar el número adecuado de iteraciones, y proceder a la optimización del diseño mediante directivas.

### 3.2.3. Determinación del número óptimo de iteraciones

A partir de los resultados de la validación, se ha calculado el error cometido tanto en el cálculo de autovalores como en el cálculo de autovectores, variando el número de iteraciones del algoritmo de Jacobi.

En la figura 3.16, se muestra el error cometido en el cálculo de autovalores, en función del número de iteraciones realizadas. Para el cálculo de los errores, se han utilizado las mismas expresiones que en el



**Figura 3.15:** Reordenamiento simplificado de las filas y las columnas de la matriz tras cada iteración

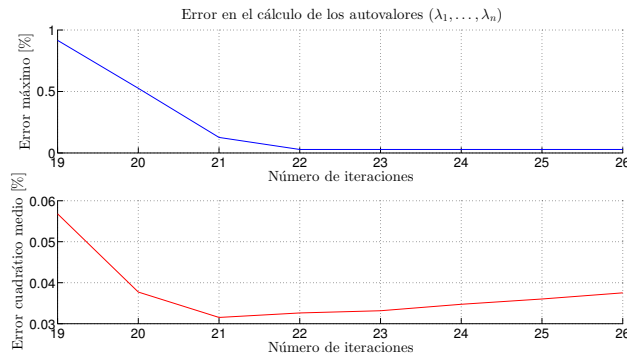
análisis del algoritmo CORDIC:

$$e_{MAX} = \max(e\%)$$

$$e\% = \frac{|X_{HLS}| - |X_{MATLAB}|}{2\pi} \cdot 100 \quad (3.7)$$

$$e_{SQRT} = \frac{1}{k^2} \sqrt{\sum_{i=1}^k e_{MAX}^2} \quad (3.8)$$

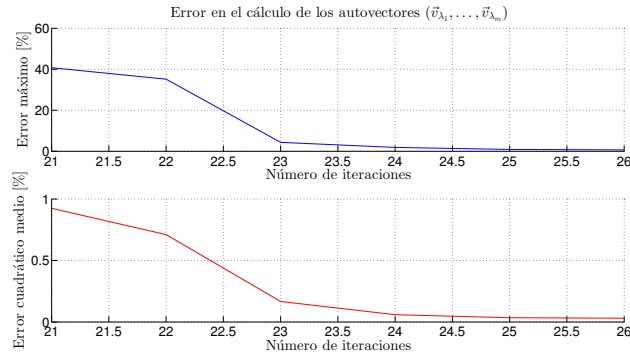
Siendo  $X$  un autovalor o componente de autovector cualquiera. Puede verse que, a partir de 22 iteraciones, el error máximo se estabiliza a un valor de 0.028 %.



**Figura 3.16:** Error cometido en el cálculo de autovalores en función del número de iteraciones del algoritmo, para matrices de dimensiones  $8 \times 8$

Por otro lado, en la figura 3.17 se muestra el error cometido al calcular los autovectores asociados a cada autovalor. En este caso, encontramos el menor error máximo a 26 iteraciones, con un valor de 0.684 %.

Puesto que, generalmente, la información más relevante la proporcionan los autovectores, se elige este último valor de cara al resto del diseño.



**Figura 3.17:** Error cometido en el cálculo de autovectores en función del número de iteraciones del algoritmo, para matrices de dimensiones  $8 \times 8$

### 3.2.4. Optimización del diseño

Ahora se van a explorar diferentes alternativas para optimizar el sistema. Tras realizar varias optimizaciones, se han encontrado dos configuraciones que ofrecen resultados similares a las arquitecturas presentadas en la sección 3.2.1.

En primer lugar, estudia el efecto que tiene el uso de CORDIC sobre el diseño. Como se justificó en la sección anterior, la optimización elegida para este sistema es una arquitectura segmentada capaz de proporcionar, tras una latencia inicial, un resultado por ciclo de reloj.

El módulo CORDIC fue concebido para operar en ambos modos de funcionamiento (rotación y vectorización), en el sistema de coordenadas circulares, pero, debido a que el número de operaciones que han de realizarse en el modo rotación, es muy superior a las que se realizan en el modo vectorización, se ha aplicado una nueva directiva de optimización que permite separarlos.

Para realizar la separación, no es necesario modificar el código fuente, siendo suficiente aplicar la directiva de optimización:

```
#pragma HLS function_instantiate variable=mode
```

Tras esto, cuando una determinada instancia hardware del módulo CORDIC solo deba funcionar en un modo, todos los recursos asociados al otro serán eliminados de forma automática. De cara al resto de optimizaciones, deberá distinguirse entre la función `cordic_scp_0` (modo rotación) y la función `cordic_scp_1` (modo vectorización).

Una vez establecido como trabajar con el módulo CORDIC, se pasa a comentar las optimizaciones del algoritmo de Jacobi. Los resultados más destacables se han encontrado actuando sobre la función completa o sobre el bucle principal, correspondiéndose con las arquitecturas paralela y serie respectivamente.

Segmentar totalmente el diseño, tiene una serie de consecuencias:

- Todos los bucles interiores se deshacen, es decir, cada iteración se implementa como un módulo hardware independiente.
- En caso necesario, los arrays se distribuyen en bancos de registros, para evitar los cuellos de botella provocados por el uso de memorias BRAM, que solo disponen de dos puertos, pudiéndose realizar únicamente 2 lecturas o escrituras por ciclo de reloj.

La primera optimización que se ha llevado a cabo, es la segmentación del diseño completo, es decir, aplicar la directiva `pipeline` a todo el código fuente, con el objetivo de conseguir la mayor velocidad de ejecución. A grandes rasgos, esto se corresponde con la arquitectura sistólica mostrada en la figura 3.12.

Deshacer el bucle principal implica que, cada llamada a la función que representa el módulo CORDIC, tendrá asociado hardware independiente. Teniendo en cuenta los recursos consumidos por este elemento, incluso en su versión reducida (implementando un solo modo de funcionamiento), esta solución requerirá un área muy grande. En la tabla 3.5 se muestran los resultados.

DSP48E	FF	LUT	$T_L/II$	CORDIC (rotación)	CORDIC (vectorización)
96(40 %)	6853(54 %)	144876(228 %)	760/32	48	1

**Tabla 3.5:** Diseño segmentado totalmente para matrices de entrada de dimensiones  $8 \times 8$ , sintetizado para el dispositivo xc7a100tcsg324-1

Como se supuso, la implementación supera con creces el número máximo de recursos disponibles en el dispositivo elegido. Para reducir el consumo de recursos, podemos optar por dos opciones, pasar directamente a la arquitectura tipo serie, o limitar el número de módulos CORDIC que puede utilizar el sintetizador para generar el hardware. En primer lugar se estudiará esta opción.

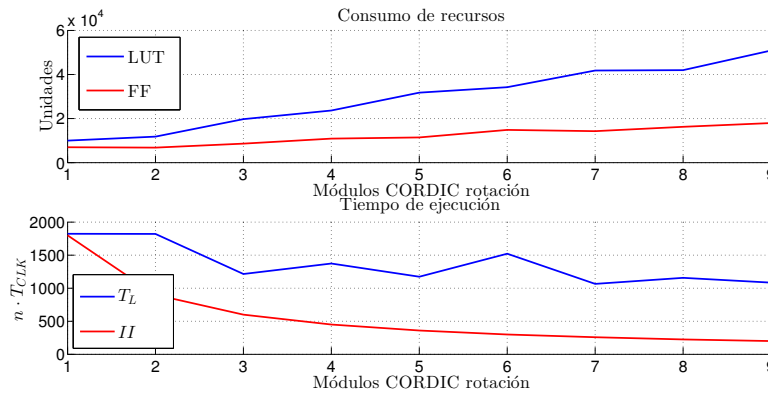
Atendiendo a los resultados anteriores (tabla 3.5), como cabría esperarse, el problema se encuentra en el excesivo número de módulos CORDIC en modo *rotacion* empleados. Para limitar su número, se recurre a la directiva de optimización:

```
set_directive_allocation -limit N -type function "cordic_svd" cordic_scp_0
```

Esta directiva limita el número de módulos CORDIC en modo rotación (`cordic_scp_0`) a  $N$  elementos en toda la función principal (`cordic_svd`).

Puesto que la lógica de control se genera de forma automática, el número de módulos CORDIC que permite alcanzar el mejor equilibrio entre el número de recursos consumidos y el tiempo de ejecución, no es conocido a priori.

Para ver como afecta el número de instancias de CORDIC en modo rotación a la solución con `pipeline` en todo el diseño, se ha realizado una simulación paramétrica variándolo entre uno y nueve. En la figura 3.18 se muestran los resultados obtenidos.



**Figura 3.18:** Latencia y recursos consumidos segmentando completamente el diseño, en función del número de módulos CORDIC, para matrices de dimensiones  $8 \times 8$

Se puede comprobar como el impacto del aumento de módulos CORDIC en modo rotación es cada vez más reducido, decreciendo el intervalo de iniciación de forma logarítmica a partir de cuatro CORDIC pero incrementándose el consumo de recursos de forma aproximadamente lineal.

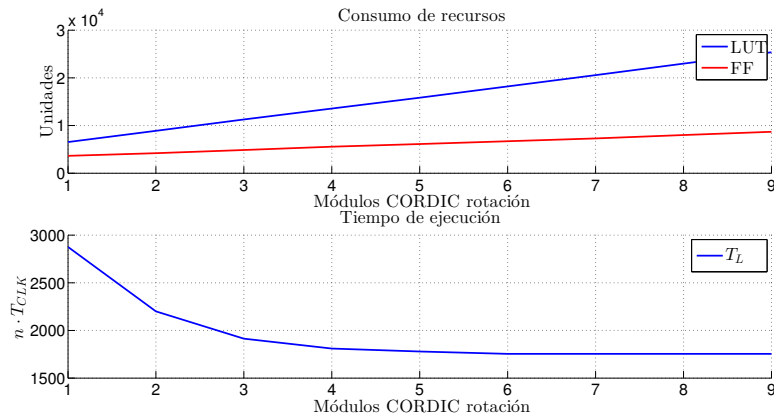
A pesar del reducido tiempo de ejecución, el consumo de recursos de esta alternativa (incluso limitando el número de módulos CORDIC), es muy elevado ya para matrices pequeñas. Además, conforme aumenta el tamaño de la matriz, si se mantiene la limitación a un número reducido de CORDIC el

sintetizador de Vivado HLS no es capaz de dar con una solución, debido a que la lógica de control se complica demasiado.

Dados los resultados, se puede concluir que, la solución con **pipeline** en el bucle principal, es adecuada cuando las matrices son pequeñas (máximo  $10 \times 10$ ) o cuando el consumo de recursos no es un problema. Para dar con un diseño más flexible, a continuación se estudia la segmentación del bucle principal.

Puesto que aplicar la directiva **pipeline** al bucle principal, que implementa las  $h$  iteraciones del algoritmo de Jacobi, implica que se desharán los bucles interiores a él (donde se realizan las rotaciones de autovalores ya autovectores), se ha recurrido a limitar el número de instancias del módulo CORDIC en modo rotación, para evitar un consumo desmesurado de recursos.

En la figura 3.19 se muestra el consumo de recursos y la latencia, cuando se aplica la directiva **pipeline** al bucle principal del algoritmo. En este caso, puesto que de cara al sistema completo no existe segmentación, el intervalo de iniciación es un ciclo mayor que la latencia, no teniendo interés para el análisis.



**Figura 3.19:** Latencia y recursos consumidos segmentando el bucle principal, en función del número de módulos CORDIC, para matrices de dimensiones  $8 \times 8$

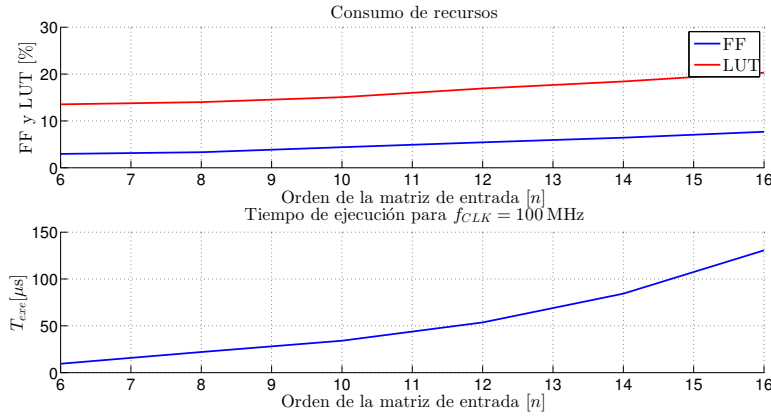
Observando los resultados puede comprobarse como, al igual que sucedía en el caso anterior, el impacto de la limitación de módulos CORDIC es cada vez menor, estabilizándose la latencia ( $T_L$ ) para 6 módulos CORDIC. Por otro lado, el consumo de recursos aumenta de forma lineal, pero es más reducido que en la alternativa totalmente segmentada, debido principalmente a que los datos internos del algoritmo se almacenan en memorias, en lugar de en bancos de registros.

Debido a que esta alternativa es más flexible, ha sido posible realizar un estudio del impacto que tiene el tamaño  $n \times n$  de la matriz de entrada sobre los resultados, realizándose una simulación paramétrica para  $n = [6, \dots, 16]$ . El número de módulos CORDIC para la simulación, se ha fijado a cuatro de forma experimental, puesto que se encontró que conforme crecía el tamaño de la matriz la latencia se estabilizaba alrededor de este punto.

En la figura 3.20 se muestran los resultados obtenidos. Puesto que se mantiene fijo el número de módulos CORDIC, el consumo de recursos se incrementa en gran medida con el tamaño de la matriz. Por otro lado, como era de esperar, la latencia crece de forma cuadrática, debido al incremento de submatrices.

### 3.2.5. Interfaz del módulo para el cálculo de autovalores y autovectores

Como ocurría en el caso del módulo CORDIC, se dispone de dos interfaces. Por un lado la interfaz software, que permite utilizar la función `cordic_svd` en otros diseños con Vivado HLS y, por otro



**Figura 3.20:** Latencia y recursos consumidos segmentando el bucle principal, en función del orden de las matrices de entrada

lado, la interfaz hardware, que permite integrar el sistema directamente en un diseño mayor, utilizando alguna herramienta del flujo de desarrollo RTL.

En el primer caso, la interfaz se corresponde con el prototipo de la función:

```
void cordic_svd( cordic_t COV_IN[MATRIX_SIZE][MATRIX_SIZE],
                 cordic_t EIGENVALUES[MATRIX_SIZE],
                 cordic_t EIGENVECTORS[MATRIX_SIZE][MATRIX_SIZE]
               );
```

Siendo `COV_IN` la matriz de la que se quieren obtener los autovalores y autovectores, `EIGENVALUES` un array que contiene los autovalores calculados y `EIGENVECTORS` una matriz que contiene los autovectores.

A partir del prototipo de la función, se genera la interfaz del diseño RTL. En este caso, todos los parámetros son arrays que, por defecto, serán implementados como una interfaz con memoria externa. Puesto que el diseño necesita recibir los datos del exterior de la manera mas rápida posible, el sintetizador supone el uso de memorias de dos puertos (DP).

En la figura 3.21 se muestran los puertos generados. Una opción interesante para simplificar la interfaz, es indicar que el array `EIGENVALUES` se implemente como una fila de la matriz `EIGENVECTORS`, mediante directivas de optimización.

Interface										
Summary										
RTL Ports	Dir	Bits	Protocol	Source Object	C Type					
COV_IN_address0	out	6	ap_memory	COV_IN	array					
COV_IN_ce0	out	1	ap_memory	COV_IN	array					
COV_IN_d0	out	18	ap_memory	COV_IN	array					
COV_IN_q0	in	18	ap_memory	COV_IN	array					
COV_IN_we0	out	1	ap_memory	COV_IN	array	EIGENVECTORS_address0	out	6	ap_memory	EIGENVECTORS array
COV_IN_address1	out	6	ap_memory	COV_IN	array	EIGENVECTORS_ce0	out	1	ap_memory	EIGENVECTORS array
COV_IN_ce1	out	1	ap_memory	COV_IN	array	EIGENVECTORS_d0	out	18	ap_memory	EIGENVECTORS array
COV_IN_d1	out	18	ap_memory	COV_IN	array	EIGENVECTORS_q0	in	18	ap_memory	EIGENVECTORS array
COV_IN_q1	in	18	ap_memory	COV_IN	array	EIGENVECTORS_we0	out	1	ap_memory	EIGENVECTORS array
COV_IN_we1	out	1	ap_memory	COV_IN	array	EIGENVECTORS_address1	out	6	ap_memory	EIGENVECTORS array
EIGENVALUES_address0	out	3	ap_memory	EIGENVALUES	array	EIGENVECTORS_ce1	out	1	ap_memory	EIGENVECTORS array
EIGENVALUES_ce0	out	1	ap_memory	EIGENVALUES	array	EIGENVECTORS_d1	out	18	ap_memory	EIGENVECTORS array
EIGENVALUES_d0	out	18	ap_memory	EIGENVALUES	array	EIGENVECTORS_q1	in	18	ap_memory	EIGENVECTORS array
EIGENVALUES_q0	in	18	ap_memory	EIGENVALUES	array	EIGENVECTORS_we1	out	1	ap_memory	EIGENVECTORS array
EIGENVALUES_we0	out	1	ap_memory	EIGENVALUES	array	ap_clk	in	1	ap_ctrl_hs	cordic_svd return value
EIGENVALUES_address1	out	3	ap_memory	EIGENVALUES	array	ap_rst	in	1	ap_ctrl_hs	cordic_svd return value
EIGENVALUES_ce1	out	1	ap_memory	EIGENVALUES	array	ap_done	out	1	ap_ctrl_hs	cordic_svd return value
EIGENVALUES_d1	out	18	ap_memory	EIGENVALUES	array	ap_start	in	1	ap_ctrl_hs	cordic_svd return value
EIGENVALUES_q1	in	18	ap_memory	EIGENVALUES	array	ap_idle	out	1	ap_ctrl_hs	cordic_svd return value
EIGENVALUES_we1	out	1	ap_memory	EIGENVALUES	array	ap_ready	out	1	ap_ctrl_hs	cordic_svd return value

Figura 3.21: Interfaz RTL del algoritmo de Jacobi diseñado en Vivado HLS





# Resultados y comparación con otras metodologías de diseño

En este capítulo se comparan los diseños tanto del algoritmo CORDIC como del algoritmo de Jacobi realizados, con sistemas implementados previamente mediante metodologías de diseño RTL.

En el caso del algoritmo CORDIC, se dispone de un subsistema codificado en VHDL sin hacer uso de *cores* propietarios, además del *IP core* proporcionado por Xilinx.

Para el estudio del algoritmo de Jacobi, se ha recurrido a los trabajos presentados en [12] y [5]. En ambas comparaciones se han utilizado como métrica los siguientes parámetros:

- Tiempo de ejecución, en términos de latencia ( $T_L$ ) y, si procede, del Intervalo de Iniciación (II).
- Frecuencia máxima de reloj.
- Recursos internos de la FPGA consumidos en la implementación.
- Error cometido en el cálculo.
- Tiempo de desarrollo.

## 4.1. Algoritmo CORDIC

Para determinar la bondad del módulo CORDIC diseñado en Vivado HLS, se va a comparar con diseños de igual funcionalidad a los que se ha tenido acceso. Concretamente, el módulo CORDIC desarrollado en [15] y un sistema basado en *cores* de Xilinx que integra tanto un módulo CORDIC que cumple la función de rotación, como uno que realiza operaciones de vectorización.

En primer lugar, se describen las características básicas de ambos diseños, presentándose los resultados obtenidos tras una implementación utilizando la herramienta ISE de Xilinx (versión 14.3). Posteriormente, se realiza la comparación con el sistema diseñado, sintetizándolo e implementándolo bajo las mismas condiciones.

### 4.1.1. Core de Xilinx

La versión utilizada es la 4.0 [16], generada mediante la herramienta *Core Generator* de Xilinx. Para que la comparación sea justa, se han seleccionado las mismas características que las utilizadas en el

diseño realizado mediante Vivado HLS.

- Ancho de palabra ( $WL$ ) de 18 bits.
- Margen de entrada para las variables  $x$  e  $y$  comprendido entre  $\pm 1$ .
- Ángulos codificados en radianes naturales, siendo posible trabajar en un intervalo de  $\pm\pi$ .
- Truncamiento.
- 17 iteraciones.
- Arquitectura paralela con el máximo (*pipeline*) posible.
- Corrección de cuadrante mediante multiplicación por  $1/K_m$  utilizando celdas DSP48E.

En la tabla 4.1 se muestran los puertos del sistema junto con una breve descripción.

Puerto	Dirección	Rotate	Translate	Descripción
X_IN	IN	✓	✓	Coordenada $x$
Y_IN	IN	✓	✓	Coordenada $y$
PHASE_IN	IN	✓	✗	Coordenada $z$
X_OUT	OUT	✓	✓	Coordenada $x$
Y_OUT	OUT	✓	✗	Coordenada $y$
PHASE_OUT	OUT	✗	✓	Coordenada $z$
ND	IN	✓	✓	Nuevos datos
CE	IN	✓	✓	Habilitación de reloj
RDY	OUT	✓	✓	Salida válida
CLK	IN	✓	✓	Señal de reloj

**Tabla 4.1:** Señales en los módulos CORDIC proporcionados por Xilinx

Del informe de *coregen*, podemos obtener la latencia para cada modo de funcionamiento:

$$T_{L,rot} = 23 \cdot T_{CLK} \qquad T_{L,vec} = 19 \cdot T_{CLK} \qquad (4.1)$$

#### 4.1.2. Diseño en VHDL

El diseño en VHDL del que se dispone, es el utilizado en [6] para la implementación del algoritmo CORDIC. A grandes rasgos, sigue una arquitectura paralela de  $n$  etapas, siendo  $n = WL$ , y utiliza la técnica de postmultiplicación para corregir el factor de escala.

Aunque es posible elegir varias alternativas de redondeo que tienen influencia sobre el error y el consumo de recursos, se ha optado por tomar la opción con truncamiento, por ser la más similar al sistema implementado en Vivado HLS.

A continuación se especifican los parámetros fundamentales.

- $n$  iteraciones en el modo rotación y  $n - 1$  iteraciones en el modo vectorización, correspondiéndose  $n$  con el ancho de palabra ( $WL$ ).
- Ancho de palabra de los datos de  $WL = 18$ bits.
- Margen de entrada de las coordenadas  $x$  e  $y$  normalizado a un intervalo de  $\pm 1$ .
- Margen de entrada para los ángulos de  $\pm\pi$  rad, codificados en radianes naturales.
- Corrección del factor de escala mediante multiplicación por  $1/K_m$ .

Los puertos disponibles en este caso son:

- **clk**: Señal de reloj.
- **reset**: Activo a nivel alto.
- **mode**: A nivel bajo activa el modo rotación.
- **load**: Se utiliza para indicar la presencia de nuevos datos.
- **done**: Se activa a nivel alto para indicar que los datos a la salida son válidos.
- **Xi, Yi, Zi**: Datos de entrada.
- **Xo, Yo, Zo, Zo2**: Datos de salida.

En este caso, a la hora de interpretar los resultados, es necesario tener en cuenta la diferencia en el número de iteraciones entre ambos modos de funcionamiento.

Para el cálculo de la latencia ( $T_L$ ), se dispone de expresiones teóricas que la relacionan con el número de iteraciones ( $n$ ). En el caso del modo rotación, la latencia  $T_{L,rot}$  se calcula como:

$$T_{L,rot} = L_i + (n - 1) \cdot T_{CLK} \quad (4.2)$$

Mientras que, en modo vectorización, el periodo de latencia  $T_{L,vec}$  se rige por la expresión:

$$T_{L,vec} = L_i + n \cdot T_{CLK} + T_{MULT} \quad (4.3)$$

Siendo  $L_i$  el tiempo que tardan en entrar los datos al sistema (registro inicial) y  $T_{MULT}$  el tiempo empleado en la corrección del factor de escala. Con la configuración seleccionada, se obtienen unos valores de:

$$T_{L,rot} = 19T_{CLK} \quad T_{L,vec} = 20T_{CLK} \quad (4.4)$$

En cuanto al error, utilizando truncamiento en los desplazamientos de bits a derechas, se obtienen los siguientes datos para un ancho de palabra de 18bits y el número de iteraciones mencionado:

$$e_{MAX,rot} = 0,18 \cdot 10^{-3} \% / 2\pi \quad e_{MAX,vec} = 0,002 \% / 2\pi \quad (4.5)$$

### 4.1.3. Comparación con el sistema diseñado mediante Vivado HLS

Analizados los diseños RTL equivalentes, se pasa a comparar sus características con el sistema diseñado en este trabajo. En la tabla 4.2 se muestran las características resumidas de los tres sistemas cuando se implementan sobre el dispositivo xc7a100csg324-1.

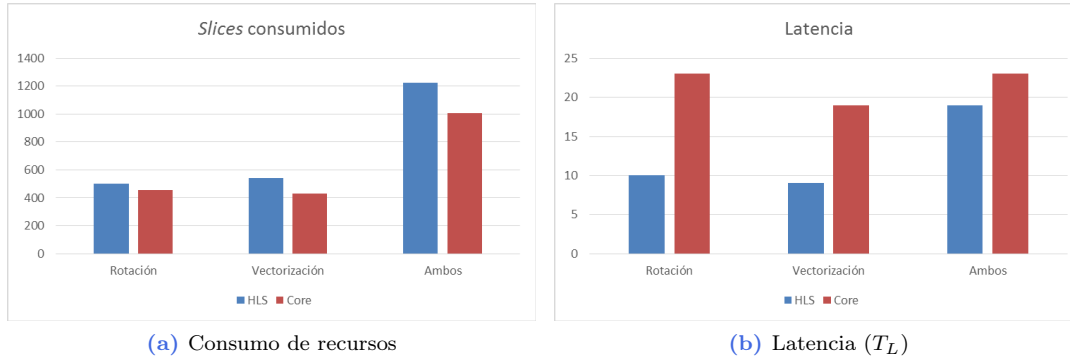
En primer lugar, se va a analizar el consumo de recursos. Si nos centramos en el módulo CORDIC creado a partir de *cores* y el diseño en Vivado HLS, además de la implementación en conjunto, es posible tener en cuenta las características de los sistemas que implementan el modo rotación y vectorización por separado.

En la figura 4.1a, se muestra una comparativa en términos de *slices* consumidos en la implementación. El sistema implementado mediante *cores* sale ganando en todas las situaciones, a pesar del consumo ligeramente menor de elementos discretos (FF y LUT) que presenta el diseño en Vivado HLS.

El problema está relacionado con la lógica de control que genera Vivado HLS, no siendo esta suficientemente óptima como para aprovechar totalmente los *slices*, que contienen cuatro LUT y ocho FF.

	HLS	VHDL	Core
Slice Reg	1780	1027	2908
Slice LUT	3353	1110	3115
Slice	1225	344	1005
DSP48E	2	2	4
$F_{CLK}$ (MHz)	194.17	379.5	211.77
$T_L$ (vec/rot)	19/19	19/20	23/27

**Tabla 4.2:** Resultados de la implementación de los tres diseños sobre el dispositivo xc7a100csg324-1 (Artix 7)



**Figura 4.1:** Comparación de recursos consumidos (slices) y latencia entre el módulo CORDIC diseñado en Vivado HLS y el sistema implementado mediante *cores* de Xilinx

Volviendo a la tabla 4.2, podemos ver que la implementación en VHDL es mucho más eficiente en cuanto a consumo de recursos se refiere, por estar optimizada a bajo nivel, pudiéndose controlar al máximo el consumo de recursos.

A partir de los datos de la tabla 4.2 y la figura 4.1b, podemos analizar la temporización de los tres sistemas. Cuando los modos de rotación y vectorización se encuentran combinados, la latencia es similar en todos los casos, siendo algo menor en el modo vectorización.

Si nos centramos en el caso donde se han separado ambos modos de funcionamiento, se puede comprobar que la latencia del sistema diseñado mediante Vivado HLS se reduce a la mitad, mientras que los *cores* de Xilinx no varían su temporización. De esto se deduce que la modularidad es un factor muy importante en Vivado HLS, puesto que permite aplicar mejores optimizaciones a los sistemas, al simplificarse la lógica de control.

Otra que conviene comentar, es la frecuencia máxima de reloj alcanzada. El diseño en VHDL alcanza el valor más alto, siendo en el sistema basado en CORES y en el diseño en Vivado HLS similar. Como ya se comentó durante la implementación, la frecuencia de reloj en Vivado HLS se establece como un objetivo en las etapas tempranas del diseño, generándose los sistemas para alcanzar un valor alrededor del periodo de reloj elegido, con un cierto margen. Si fuera necesaria una mayor frecuencia de operación, podría conseguirse a costa de una mayor latencia.

Pasando al error cometido en el cálculo, es similar en todos los sistemas, puesto que el sistema de codificación y la estrategia a la hora de tratar los desbordamientos en las operaciones internas es igual.

Finalmente, a pesar de ser un parámetro subjetivo, se comenta a grandes rasgos el tiempo de desarrollo para el sistema diseñado en este trabajo y el módulo CORDIC codificado en VHDL. El módulo CORDIC fue, tras el proceso de aprendizaje, la primera tarea que se llevo a cabo, tomando su análisis e implementación en Vivado HLS aproximadamente un mes.

Por otro lado, el diseño codificado en VHDL, se concibió como parte de una tesis doctoral, llevando aproximadamente cuatro meses y medio a un ingeniero que ya tenía experiencia en la codificación en VHDL y el diseño digital. Esto implica que la validación e implementación de un sistema de igual funcionalidad, ha llevado un tiempo cuatro veces menor en Vivado HLS que en VHDL, a cambio de un consumo de recursos aproximadamente cuatro veces mayor.

## 4.2. Algoritmo de Jacobi

El análisis de los resultados obtenidos del diseño e implementación del método de Jacobi, se ha llevado a cabo de forma similar al presentado en la sección anterior para el algoritmo CORDIC.

En este caso, se parte de un diseño en VHDL, basado en el módulo CORDIC presentado y de un sistema realizado en XSG, ambos diseñados para trabajar con matrices de dimensiones  $8 \times 8$ .

Puesto que prácticamente toda la carga computacional del algoritmo de Jacobi recae sobre los módulos CORDIC, es de esperar que las prestaciones de estos sistemas estén muy relacionadas con las del módulo CORDIC utilizado para implementar el sistema.

### 4.2.1. Cálculo de autovalores y autovectores en XSG

En XSG, los sistemas se implementan en la herramienta Simulink de Matlab, realizando diagramas de bloques a partir de elementos discretos (registros, puertas lógicas) y *cores* (contadores, memorias, celdas DSP48E, módulos CORDIC,...). A partir del diagrama de bloques, se genera código RTL en VHDL o Verilog, sintetizable e implementable en los dispositivos de Xilinx.

El trabajo que se analiza [7], implementa un sistema equivalente al presentado en [6], utilizando el subsistema *sincos* de Xilinx como base del módulo CORDIC, en lugar de los *cores* de rotación y vectorización, para reducir el consumo de recursos.

El sistema presenta una latencia de:

$$T_L = 3717 \cdot T_{CLK} \quad (4.6)$$

Por otro lado, el error porcentual cometido en el cálculo de autovalores es:

$$e_{MAX,\lambda} = 0,9 \% \quad e_{SQRT,\lambda} = 0,5 \% \quad (4.7)$$

### 4.2.2. Cálculo de autovalores y autovectores codificado en VHDL sin *cores*

El diseño en VHDL se corresponde con la arquitectura presentada en [12]. Este sistema fue diseñado para trabajar con matrices de dimensiones  $8 \times 8$ , como parte de un sistema de visión artificial que implementaba la técnica estadística PCA, donde se trabaja con matrices de covarianza.

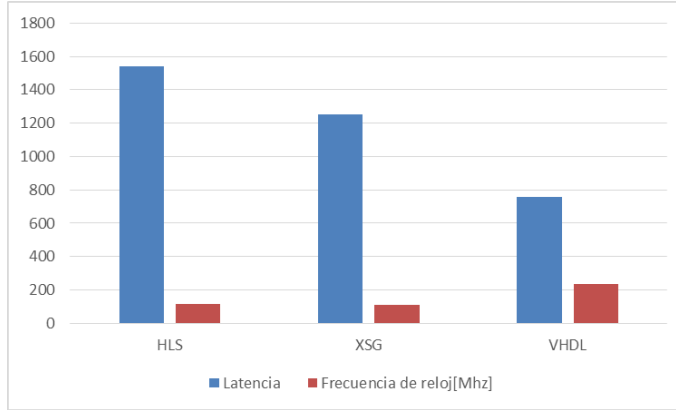
Puesto que la gran mayoría de los multiplicadores de la FPGA debían ser utilizados en otros subsistemas, fue concebido para hacer el menor uso posible de los mismos, filosofía que se ha seguido también en este trabajo.

El tiempo total ( $T_L$ ) empleado para el cálculo de autovalores y autovectores de una matriz dada, es de:

$$T_L = 1591 \cdot T_{CLK} \quad (4.8)$$

Mientras que, el error porcentual máximo cometido en el cálculo de los autovalores es:

$$e_{MAX,\lambda} = 0,006 \% \quad e_{SQRT,\lambda} = 0,005 \% \quad (4.9)$$



**Figura 4.2:** Comparativa de latencia y frecuencia de reloj máxima para el algoritmo de Jacobi implementado mediante diferentes metodologías de diseño.

#### 4.2.3. Comparación con el sistema diseñado en Vivado HLS

De las alternativas de optimización exploradas en Vivado HLS, para llevar a cabo la comparación se ha decidido tomar la que se asemeja a la arquitectura serie, en su versión con cuatro módulos CORDIC en modo rotación, y uno en modo vectorización.

En primer lugar, se han sintetizado e implementado los tres sistemas en ISE para el dispositivo xc7a100tcs324-1. En la tabla 4.3 se muestra un resumen de los resultados obtenidos.

	HLS	XSG	VHDL
Slices	3287	1255	756
	20.7 %	7.9 %	4.7 %
$T_L[nT_{CLK}]$	1543	3717	1591
$f_{CLK}[\text{MHz}]$	116.18	111.36	236.17

**Tabla 4.3:** Resultados de la implementación del sistema realizado mediante diferentes metodologías de diseño, en el dispositivo xc7a100tcs324-1.

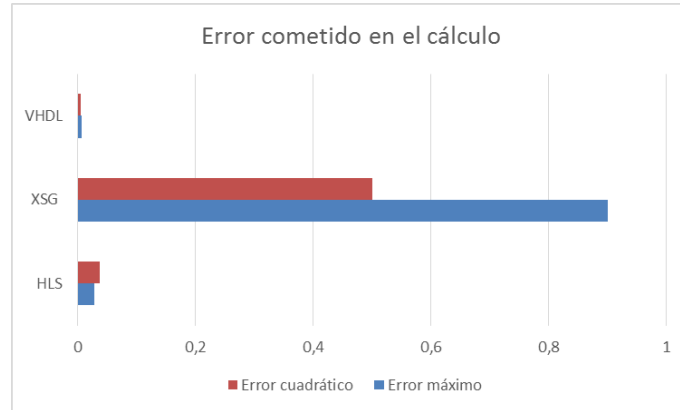
El sistema codificado en VHDL consigue el menor consumo de recursos, seguido por el diseño en XSG, debido a su concepción en bajo nivel, que permite un mejor aprovechamiento de los recursos internos de la FPGA. A la vista de los resultados de la comparación de los diferentes módulos CORDIC en la sección anterior, esta era la situación esperada.

En la figura 4.2, se muestra una comparativa de la latencia y la frecuencia máxima de reloj alcanzable por cada sistema. En este caso, es el sistema realizado mediante síntesis de alto nivel el que presenta unas mejores prestaciones.

El motivo de que los diseños en Vivado HLS puedan superar fácilmente a otras alternativas diseñadas mediante metodologías RTL en tiempo de ejecución, es la optimización que realiza el sintetizador para introducir en cada ciclo de reloj el mayor número posible de operaciones, teniendo en cuenta las características del dispositivo elegido y la frecuencia de reloj objetivo.

Esto pudo verse ya en la implementación del módulo CORDIC cuando se separan ambos modos de funcionamiento, donde se podían introducir dos iteraciones del algoritmo en cada etapa del *pipeline*.

Cuando se trabaja a bajo nivel, es el diseñador el que debe decidir que operaciones se ejecutan en cada ciclo de reloj, no siendo sencillo combinar el mayor número de operaciones por ciclo de reloj con el diseño de la lógica de control, recurriéndose a criterios experimentales para decidir en que punto se introducen los registros del *pipeline*.



**Figura 4.3:** Error porcentual máximo cometido en el cálculo de autovalores y autovectores por los tres sistemas bajo análisis.

Se puede observar también una gran diferencia entre la latencia de los diseños en Vivado HLS y VHDL respecto al sistema realizado mediante XSG. Esto se debe a que el sistema en XSG está concebido a partir del diseño codificado en VHDL, con la salvedad del uso de dos señales de reloj, a las que denominaremos  $f_1$  y  $f_2$  con  $f_2 = 2f_1$ .

En el sistema codificado en VHDL presentado en [6], se utilizaba una señal de reloj  $f_2$  del doble de frecuencia que la señal principal del sistema ( $f_1$ ) para, en cada ciclo de reloj, poder leer cuatro datos de una memoria DP y colocarlos a la entrada de dos módulos CORDIC, en la realización de las operaciones de rotación.

En cuanto al error cometido, en la figura 4.3 se muestra una comparativa del error porcentual máximo y el error cuadrático medio cometido en el cálculo de los autovalores. El sistema codificado en VHDL presenta un error despreciable frente a los otros dos, siendo los resultados ofrecidos por la solución en Vivado HLS mucho mejores que los arrojados por el sistema diseñado mediante XSG.

#### 4.2.4. Comparación con la función `svd` de la librería de álgebra lineal de Vivado HLS

Finalmente, se va a realizar una comparación del sistema diseñado en Vivado HLS con la función proporcionada por Xilinx en la librería de álgebra lineal de Vivado HLS, introducida en la versión 2014.1.

Esta función implementa el algoritmo de Jacobi para el cálculo de autovalores y autovectores en matrices cuadradas y simétricas, por división en submatrices de dimensiones  $2 \times 2$ . La codificación utilizada es *coma flotante de precisión simple* y no se hace uso del algoritmo CORDIC para resolver las operaciones.

Realizando el proceso de síntesis en Vivado HLS sobre el dispositivo xc7a100csg324-1, se obtienen los resultados mostrados en la figura 4.4. Antes de continuar con la comparación, hay que tener en cuenta que la implementación en coma flotante implica el uso de *cores* de coma flotante, que implican un incremento en el consumo de recursos y tiempo de ejecución de las operaciones. Por otro lado, la ventaja que aporta esta codificación son resultados muy exactos, cuyo error proviene en mayor medida de la naturaleza iterativa del algoritmo de Jacobi que de los errores de truncamiento.

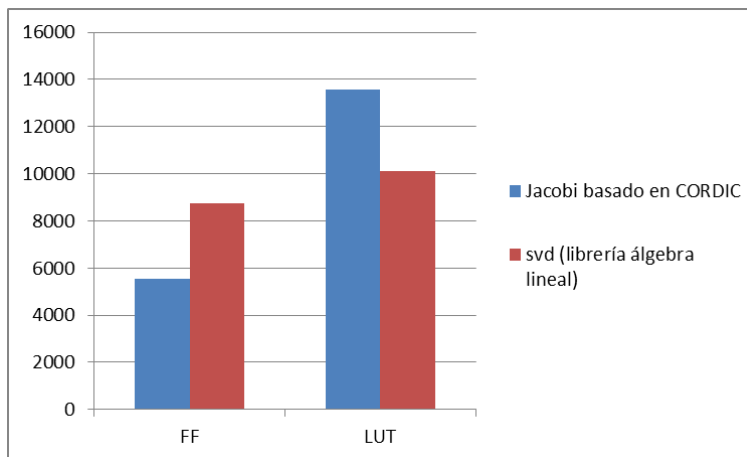
Si nos centramos en primer lugar en el consumo de recursos, respecto a la solución utilizada para comparar con el resto de alternativas de diseño en las secciones anteriores (cuatro módulos CORDIC en modo rotación y el bucle principal), encontramos que el consumo de DSP48E es de 48 frente a 8, es decir, seis veces mayor. Dependiendo de la aplicación, el uso de los multiplicadores internos de la

Utilization Estimates					Latency (clock cycles)				
Summary					Summary				
Name	BRAM_18K	DSP48E	FF	LUT	Latency		Interval		
Expression	-	-	0	96	min	max	min	max	Type
FIFO	-	-	-	-	742	122822	743	122823	none
Instance	3	58	8655	9976					
Memory	4	-	0	0					
Multiplexer	-	-	-	56					
Register	-	-	101	-					
Total	7	58	8756	10128					
Available	270	240	126800	63400					
Utilization (%)	2	24	6	15					

**Figura 4.4:** Resultados obtenidos tras la síntesis en Vivado HLS de la función `svd` de la librería de álgebra lineal de Xilinx para matrices de dimensiones  $8 \times 8$

FPGA puede no ser posible si son necesarios para otra operación, como la multiplicación de matrices, muy habitual en aplicaciones que utilizan el cálculo de autovalores y autovectores (p.ej. la técnica PCA).

En cuanto al consumo de biestables (FF) y LUT, en la figura se muestra una comparativa, encontrándose ambos en rangos similares.



**Figura 4.5:** Comparación del consumo de recursos entre la función `svd` de la biblioteca de álgebra lineal de Vivado HLS y el sistema con *pipeline* en el bucle principal y cuatro módulos CORDIC en modo rotación, para matrices de dimensiones  $8 \times 8$

Analizando ahora la temporización, mientras que en el sistema implementado en Vivado HLS la latencia ( $T_L$ ) es constante e igual a **1810** ciclos de reloj, en la implementación en coma flotante varía entre un intervalo de  $[742, 122822]$  ciclos de reloj, dependiendo de las características de la matriz de entrada, y no pudiendo ser determinada de antemano.



# Conclusiones y trabajos futuros

## 5.1. Conclusiones

En este trabajo se ha desarrollado una implementación del algoritmo de Jacobi para el cálculo de autovalores y autovectores en FPGA, mediante síntesis de alto nivel.

Esta metodología de diseño, cambia totalmente la forma de trabajar respecto al diseño de sistemas mediante descripciones RTL realizadas, por ejemplo, en VHDL. Muchos conceptos se redefinen y es necesario adaptarse a ellos, además de cambiar la forma de abordar la planificación de los diseños.

Para poder llevar a cabo los objetivos propuestos en el trabajo, se pasó por una fase de aprendizaje, reflejada en parte en la *Introducción a Vivado HLS* del apéndice A, que permitió formar una primera opinión sobre la síntesis de alto nivel y aprender el manejo de la herramienta.

En esta etapa temprana del trabajo, se llegó a la conclusión de que utilizar Vivado HLS para implementar un sistema completo, al menos en un único diseño, no era una opción viable, pero tenía un gran potencial como herramienta para diseñar subsistemas que implementen algoritmos matemáticos que después serán integrados en el flujo de desarrollo RTL.

Como ya se comentó en la introducción, la forma de implementar algoritmos matemáticos mediante metodologías RTL no es intuitiva, requiriéndose una etapa de estudio y adaptación muy dependiente de sus características. Esto implica grandes tiempos de desarrollo, necesidad de un profundo conocimiento de los algoritmos y experiencia en el diseño digital.

Tras la experiencia adquirida en el desarrollo del trabajo, se puede concluir que la síntesis de alto nivel puede solucionar estos problemas, siempre que sea admisible pagar un coste en otros aspectos del diseño. Los algoritmos que se más se prestan a ser implementados de esta forma son aquellos que, a grandes rasgos, tienen las siguientes características comunes:

- Carga computacional suficientemente elevada como para que la implementación en sistemas basados en microprocesador no cumplan las restricciones temporales de la aplicación.
- Características internas, en cuanto a dependencias de datos se refiere, que permitan paralelizar la ejecución.
- Conjuntos de operaciones relativamente simples pero repetitivas.

Durante la implementación del algoritmo de Jacobi se pudo ver que, partiendo de una descripción algorítmica sin apenas modificaciones respecto al estudio teórico, se llegó a diversas implementaciones con características muy dispares, alcanzando a grandes niveles de paralelismo sin la necesidad de segmentar y planificar las tareas de forma manual.

Continuando con este último punto, se ha encontrado especialmente útil la posibilidad de iterar sobre los diseños, obteniendo diferentes implementaciones hardware a partir de una misma descripción, estudiando las dependencias de datos y el uso de recursos para localizar los puntos en los que el diseño podía ser optimizado.

Esto último implica que las decisiones iniciales de diseño, no son un punto de no retorno, siendo posible explorar diferentes arquitecturas hardware sin tener que volver a codificar todo el sistema. Por supuesto, todas estas ventajas vienen acompañadas de una serie de inconvenientes.

Las principales limitaciones de la síntesis de alto nivel que se han encontrado durante la realización de este trabajo, pasan por el mayor consumo de recursos de los sistemas y la generación de algunas soluciones válidas en apariencia pero no implementables en el dispositivo real. La imposibilidad de trabajar a bajo nivel, no se entiende como una desventaja, puesto que no es el objetivo de la herramienta.

En cuanto al primer punto, se ha llegado a la conclusión de que la lógica de control que genera Vivado HLS para llevar a cabo todas las operaciones de temporización, es excesivamente complicada y, en ocasiones, muy redundante y poco optimizada, encontrándose diferencias en torno al 40 % entre las estimaciones de recursos de Vivado HLS y el uso real de los mismos tras el proceso de síntesis e implementación mediante ISE.

Por otro lado, el problema de las soluciones no implementables está directamente relacionado con la generación de la lógica de control que realiza Vivado HLS, permitiéndose en ocasiones llevar a cabo optimizaciones sobre el diseño, que en realidad no son viables en la práctica.

Como ejemplo de este problema, cuando se estudió la limitación del número de módulos CORDIC en la arquitectura paralela del algoritmo de Jacobi (apartado 3.2.4), se encontró que la opción con dos subsistemas CORDIC daba unos resultados teóricos muy competitivos, pero, al intentar realizar la implementación en ISE, el proceso de *Place & Route* no podía ser completado con éxito.

También se ha podido observar este problema al separar los subsistemas de rotación y vectorización del algoritmo CORDIC, mejorando la latencia ( $T_L$ ) en un 50 % respecto al sistema que permitía elegir el modo de funcionamiento mediante una señal de control.

## 5.2. Trabajos futuros

Puesto que en este trabajo se han abordado dos líneas diferentes: la síntesis de alto nivel como metodología de diseño hardware y el cálculo de autovalores y autovectores en hardware reconfigurable, sería posible continuar el trabajo desde diferentes puntos.

Por la proliferación que han tenido en los últimos años las herramientas de síntesis de alto nivel, denominadas de tercera generación, la evaluación de las mismas mediante la implementación de diseños ya conocidos es un campo por explorar.

Además de Vivado HLS, encontramos otras herramientas que siguen la misma metodología (diseñar mediante lenguajes de programación de alto nivel), como HDL coder de MATLAB, ImpulseC, de Impulse o Catapult C de Mentor Graphics. La implementación de los sistemas llevados a cabo en este trabajo en estas herramientas, sería interesante para comprobar el nivel de optimización que se puede alcanzar y la dificultad a la hora de trasladar un diseño realizado en Vivado HLS a otra herramienta similar.

Por otro lado, el cálculo de autovalores y autovectores, es un campo de las matemáticas que ha ido ganando cada vez más importancia dentro de la ciencia y la ingeniería, por su utilidad para analizar grandes cantidades de datos.

Como continuación directa de este trabajo, se plantea la modificación del algoritmo para el cálculo de autovalores y autovectores en matrices de mayor tamaño (p.ej.  $512 \times 512$ ), dimensiones habituales en aplicaciones como la visión artificial. Una posibilidad, sería realizar una división en submatrices de un

tamaño múltiplo de dos, como  $16 \times 16$ , resolviendo el problema mediante una estrategia similar a la división en matrices  $2 \times 2$  utilizando el sistema desarrollado en este trabajo.

Además de la versión del algoritmo de Jacobi utilizada en este trabajo, existen modificaciones que siguen metodologías similares tratando de reducir el número de operaciones realizadas. Por ejemplo, en [14] se trabaja con módulos CORDIC que adaptan el número de iteraciones (y por tanto la precisión), conforme avanza el cálculo, reduciendo el tiempo de ejecución.

Otra posibilidad, pasa por realizar rotaciones únicas en el cálculo de autovalores, de forma similar a lo que ocurre en el cálculo de autovectores, pudiéndose implementar una arquitectura sistólica similar a la contemplada en este trabajo, también diseñada por brent [17].

A pesar de que los métodos basados en el algoritmo de Jacobi son los más empleados cuando el cálculo ha de realizarse en sistemas hardware, existen numerosas soluciones al problema de los autovalores y autovectores en matrices simétricas.

Dada la facilidad para implementar algoritmos matemáticos en las herramientas de síntesis de alto nivel como Vivado HLS, una posible línea de investigación pasa por la evaluación de otros métodos, como el algoritmo QR o el método de la bisección.

Ambos se fundamentan en la tridiagonalización de la matriz de entrada, recurriéndose generalmente a reflexiones de Householder en el primer caso y al algoritmo de Lanczos en el segundo, iterándose sobre la matriz resultante. Por su naturaleza, se prestan al uso de codificación en coma flotante, habiendo sido implementados tradicionalmente sobre procesadores de propósito general.



## Parte III

# Pliego de condiciones



# Capítulo 6

## Pliego de condiciones

En este capítulo se presentan los recursos software y hardware que son necesarios para llevar a cabo el trabajo.

### 6.1. Requisitos hardware

- Ordenador personal (portatil) Acer Aspire E1-572G
  - Procesador intel core i7-4500U
  - 8Gb de memoria RAM

### 6.2. Requisitos software

Requisitos generales:

- Sistema operativo Windows 8.1.
- Editor de textos Emacs.

Para la realización del trabajo:

- Vivado Suite 2014.1
- ISE Design Suite 14.3
- MATLAB 2013a junto con la *toolbox* de diseño en coma fija (Fixed-Point Designer).

Para la redacción de la memoria:

- Inkscape (edición de gráficos vectoriales).
- Sumatra PDF (lector de archivos pdf).
- TeXMaker + TeXlive (Entorno de desarrollo y distribución LaTeX)





Parte IV

Presupuesto



# Capítulo 7

## Presupuesto

El coste de los recursos se divide en los siguientes apartados: recursos software (tabla 7.1), recursos hardware (tabla 7.2) y mano de obra (tabla 7.3), y material fungible y otros costes (7.4).

Recursos software				
Concepto	Coste	Amortización (años)	Tiempo de uso (meses)	Total
MATLAB 2013a	2000€	4	5	208.33€
Fixed-Point Designer (MATLAB)	2500€	4	5	260.42€
Vivado System Edition (con ISE)	3517€	4	5	366.35€
Vivado HLS	1463€	4	5	152.40€
Windows 8 Pro	279€	5	5	23.25€
Sumatra PDF	0€	-	5	0€
Inkscape	0€	-	5	0€
TeXMaker	0€	-	5	0€
TeXLive	0€	-	5	0€
Total				1010.75€

Tabla 7.1: Coste de los recursos software

Recursos hardware				
Concepto	Coste	Amortización (años)	Tiempo de uso (meses)	Total
Ordenador portatil Acer Aspire E1-572E	700 €	5	5	58.33
Total				700 €

Tabla 7.2: Coste de los recursos hardware

Mano de obra			
Concepto	Coste por hora	Horas	Coste
Ejecución	30€/h	225h	6750€
Redacción	12€/h	75h	900€
Total			7650€

Tabla 7.3: Coste de la mano de obra

A partir de los datos expuestos, se procede a calcular el coste total del proyecto. El *presupuesto de*

Material fungible y otros costes	
Concepto	Coste
DVD-RW	1.5€
Impresión y encuadernación de los libros	80€
Total	81.5€

**Tabla 7.4:** Coste del material fungible y los libros

*ejecución por contrata* se obtiene mediante la siguiente fórmula:

$$PEC = CM + b\% \cdot CM$$

Donde:

**PEC** es el Presupuesto de Ejecución por Contrata.

**CM** es el coste total de ejecución material del proyecto, calculado como la suma del coste de los recursos software, el coste de los recursos hardware, el material fungible y la mano de obra.

$b\%$  es el porcentaje del coste material del proyecto tomado como beneficio industrial.

El coste total de ejecución material toma un valor de:

$$CM = 1010,75 + 58,33 + 7650 + 81,5 = 8800,58€$$

Suponiendo un 10 % de beneficio industrial sobre el coste del proyecto, el presupuesto de ejecución por contrata se calcula como:

$$PEC = 8800,58 + 0,1 \cdot 8800,58 = 9680,64€$$

A partir del Presupuesto de Ejecución por Contrata (PEC) se calculan los Honorarios Facultativos (HF), establecidos como un 7 % del PEC cuando este es menor a 30,050,61€.

$$HF = 0,07 \cdot PEC = 0,07 \cdot 9680,638 = 677,64€$$

El Coste Total (CT) del proyecto será la suma del presupuesto de ejecución por contrata y los honorarios facultativos, a los que deberá añadirse el 21 % de IVA. El coste antes de impuestos es:

$$CT_{AI} = PEC + HF = 9680,64 + 677,64 = 10358,28€$$

Siendo el coste total tras aplicar el impuesto correspondiente:

$$CT = 1,21 \cdot CT_{AI} = 1,21 \cdot 10358,28 = 12533,52€$$

El importe final del proyecto asciende a la cantidad de **DOCE MIL QUINIENTOS TREINTA Y TRES EUROS CON CINCUENTA Y DOS CÉNTIMOS**.

Parte V

Apéndices



## Introducción a Vivado HLS

En este apéndice se da una introducción a la síntesis de alto nivel con Vivado HLS. En la sección [A.1](#) se da una visión general de la síntesis de alto nivel y, en la sección [A.2](#), se describen las partes más importantes del entorno de trabajo.

La sección [A.3](#) trata la estructura de los diseños en Vivado HLS junto con las diferencias y similitudes con los lenguajes de descripción RTL. El resto de secciones describen los elementos disponibles para llevar a cabo los diseños.

En [A.4](#) se trata la especificación de interfaces y su relación con las entidades de los lenguajes HDL<sup>1</sup>. Finalmente, en la sección [A.5](#), se presentan los elementos del lenguaje C más relevantes en la síntesis de alto nivel y las diferentes posibilidades a la hora de optimizar los diseños.

### A.1. Síntesis de alto nivel

El objetivo principal de la síntesis de alto nivel, es proporcionar al diseñador una manera rápida de implementar algoritmos en hardware, partiendo de una especificación software. En el caso de Vivado HLS, la conversión se realiza a partir de una función C, C++ o System C.

Para convertir una descripción software en un módulo hardware equivalente, se asocian los elementos del lenguaje de programación utilizado, a elementos propios del diseño digital y se imponen ciertas restricciones semánticas en el bloque de código a sintetizar, denominado *top function*.

En cuanto a la metodología de diseño, Vivado HLS permite la validación del algoritmo tanto a nivel software (para verificar su corrección) como a nivel hardware (para comprobar que la transformación a sido adecuada). Por otro lado, también es posible aplicar una serie de optimizaciones sobre un mismo diseño, para buscar el equilibrio deseado entre tiempo de ejecución y recursos consumidos.

Una vez se considera que el sistema generado cumple las especificaciones fijadas, puede ser exportado para integrarlo en otro diseño. En este punto se ofrecen diferentes alternativas.

- Obtener directamente el código RTL en el lenguaje deseado: VHDL, Verilog o System C.
- Crear un *core* para su uso en Vivado Suite o ISE.
- Crear un *core* para su uso en un sistema basado en procesador, por ejemplo en la plataforma Zynq.
- Generar un bloque para utilizarlo en XSG como parte de un diseño basado en modelos.

---

<sup>1</sup>Hardware Description Language

## A.2. Instalación y entorno de trabajo

Vivado HLS viene incluido en la *Vivado Suite* de Xilinx y se licencia de forma independiente al resto de herramientas.

El entorno de trabajo está basado en el IDE<sup>2</sup> Eclipse y adaptado a las particularidades de la síntesis de alto nivel. Dentro del mismo se dispone de todo lo necesario para llevar a cabo el diseño y test de módulos hardware completos.

La metodología de trabajo está basada en proyectos, siendo posible generar diferentes implementaciones hardware a partir de un mismo proyecto. La forma de trabajar con los proyectos puede ser manual, a través de la interfaz gráfica, o automática, mediante el uso de scripts `tcl`.

Dentro de un proyecto podemos diferenciar varias partes:

- Características generales del diseño (FPGA seleccionada, frecuencia de reloj objetivo, ...)
- Código fuente:
  - Relacionado con la implementación *top function* y todas las funciones de menor jerarquía utilizadas.
  - Relacionado con las pruebas (*test bench*).
- Soluciones, englobando lo anterior junto con una serie de directivas especificadas por el usuario, para generar una implementación del diseño.

Dependiendo de la tarea que se está realizando, se puede acceder a tres perspectivas diferentes: síntesis, depuración y análisis.

**Síntesis (*Synthesis*):** es la perspectiva principal, donde se desarrolla el código y se aplican las diferentes directivas de optimización.

**Depuración (*Debug*):** es una perspectiva de depuración de código C/C++, igual que la que se utiliza en los entornos de desarrollo software.

**Análisis (*Analysis*):** en esta perspectiva pueden estudiarse las características de una implementación en cuanto temporización y recursos, y el comportamiento del hardware generado.

En la figura A.1 se muestra la perspectiva de síntesis con las zonas más importantes destacadas. Encontramos cuatro zonas diferenciadas.

1. En la esquina superior izquierda aparecen los controles para moverse entre las diferentes perspectivas.
2. En la parte superior, destacados en magenta, están los ejecutables de las diferentes herramientas, de izquierda a derecha: creación de nuevas soluciones, compilador, sintetizador, simulación hardware y exportación.
3. A la izquierda aparece la estructura del proyecto
  - Código fuente para la síntesis (*top function* y todas las funciones con menor jerarquía).
  - Código fuente para las pruebas (función *main* y todas la de mayor jerarquía a la *top function*).
  - Soluciones, cada una representa una implementación del mismo código fuente con diferentes directivas (interfaz, optimización, ...).
  - Dentro de las soluciones aparecen dos scripts `*.tcl`, en uno de ellos se encuentran las directivas de optimización y el otro permite crear el proyecto y la solución de forma automática.

---

<sup>2</sup>Integrated Development Environment



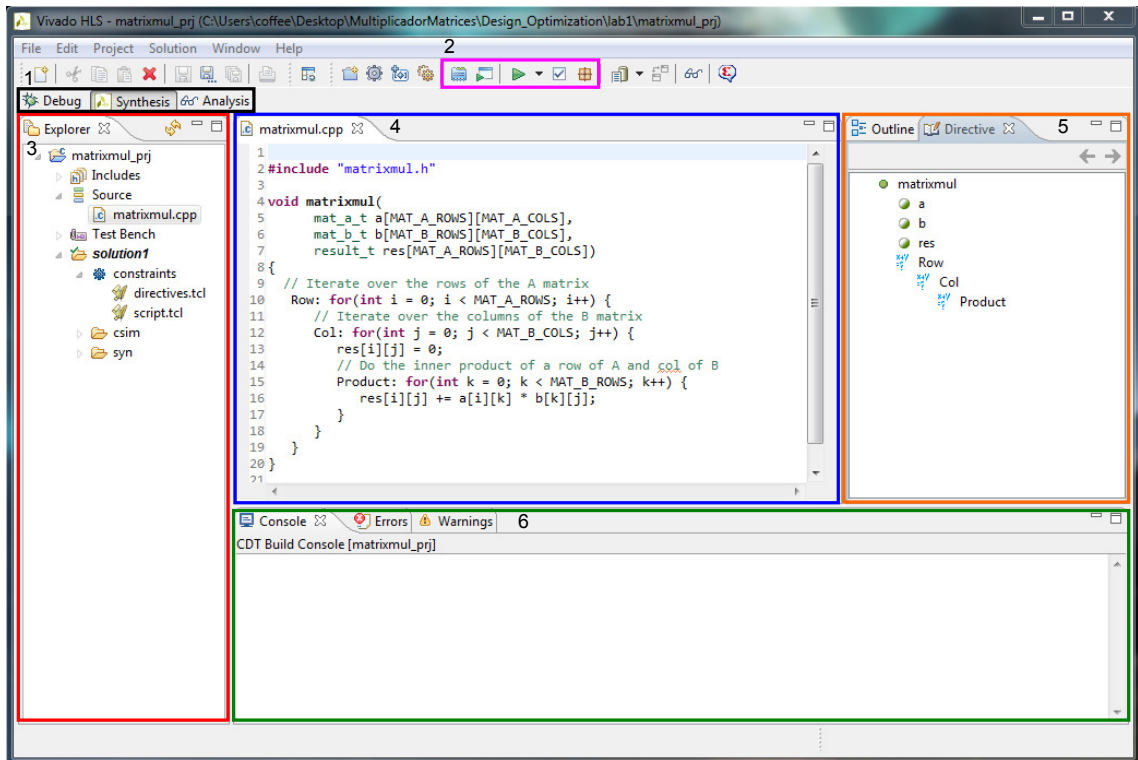


Figura A.1: Perspectiva de Síntesis del entorno Vivado HLS

4. En el centro encontramos el editor.
5. A la derecha se puede ver la pestaña de directivas, donde es posible aplicar optimizaciones sobre una determinada solución o sobre el proyecto.
6. En la parte inferior se localizan: la consola, donde aparecen los mensajes de las diferentes herramientas a las que llama el entorno (compilador, sintetizador y simulador RTL); y la entrada/-salida estándar, en caso de utilizarse.

La perspectiva de análisis permite estudiar las características del diseño hardware una vez realizada la implementación, pudiendo ver el flujo de ejecución generado, los recursos consumidos y las características temporales. En la figura A.2 puede verse una captura de pantalla de la misma. A continuación se destacan los elementos más importantes:

1. Jerarquía (*Module Hierarchy*): muestra la *top function* y todas las funciones utilizadas dentro de la misma.
2. Rendimiento (*Performance Profile*): muestra la cantidad de ciclos de reloj que tarda el diseño en procesar el primer set de datos de entrada (*Latency*) y el número de ciclos de reloj necesarios para aceptar datos nuevos a la entrada (*Initiation Interval*).
3. Estimación de recursos consumidos (*Resource Profile*): muestra los recursos internos de la FPGA de forma detallada los recursos necesarios para implementar el módulo generado.

### A.3. Estructura de un diseño en Vivado HLS

A partir de ahora, salvo que se indique lo contrario, el desarrollo se centrará en el lenguaje de programación C, puesto que ha sido el utilizado en este trabajo.

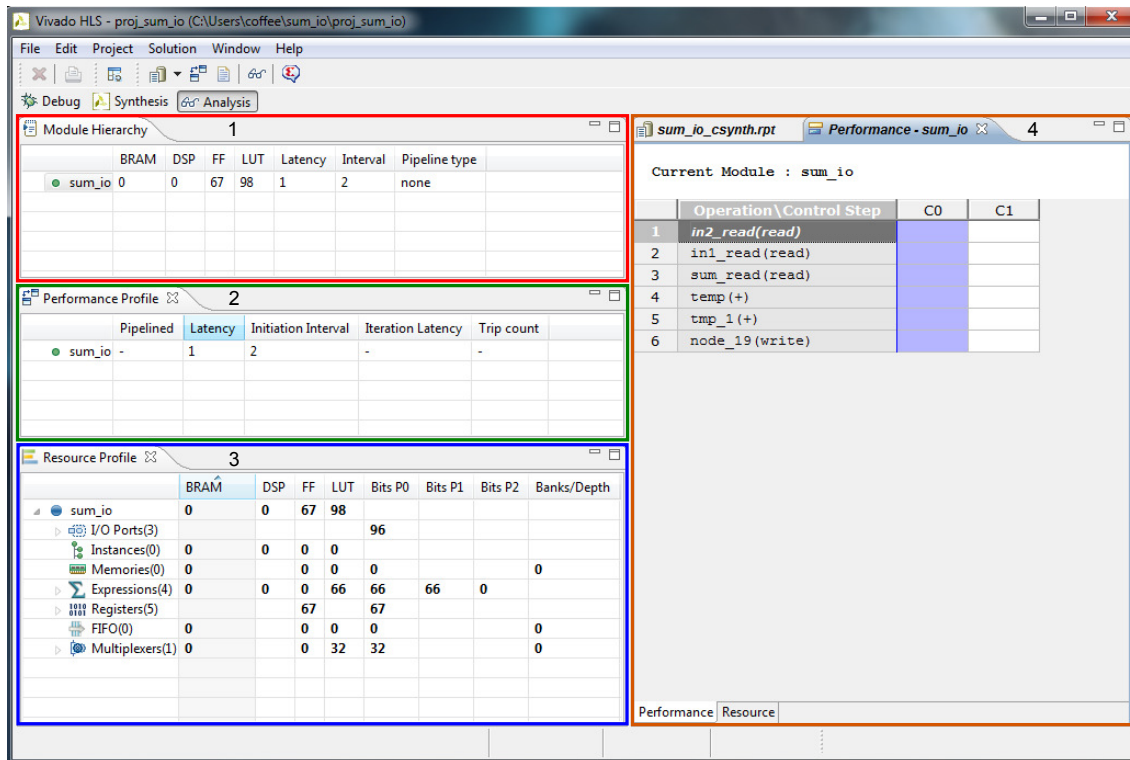


Figura A.2: Perspectiva de Análisis del entorno Vivado HLS

La estructura de una descripción RTL, realizada mediante un lenguaje como Verilog o VHDL, se compone de una serie de módulos de diferente jerarquía. El módulo de mayor jerarquía suele ser utilizado para realizar pruebas sobre el inmediatamente inferior, que es el que posteriormente será sintetizado e implementado en la FPGA, o integrado en otro diseño.

Un módulo hardware escrito, por ejemplo, en VHDL, consta de dos partes: Una entidad, donde se establecen los puertos del diseño; y una arquitectura, donde se describe su funcionalidad..

En Vivado HLS, puesto que se utilizan lenguajes de programación de alto nivel, la modularidad de los diseños se consigue mediante el uso de funciones y la forma de utilizarlas establece el orden jerárquico. En el lenguaje C y sus derivados, la función *main* es la que tiene una mayor jerarquía.

La función *main* será por tanto una zona sin restricciones en el lenguaje, donde se realicen pruebas sobre otra función, que será transformada posteriormente en una descripción RTL. Esta función se denomina *top function*.

En la *top function* encontramos dos zonas diferenciadas, los argumentos y el cuerpo. Los argumentos serán tomados como base para generar los puertos del diseño, junto con una serie de directivas dadas por el diseñador (entidad) mientras que, en el cuerpo de la función, se describe la funcionalidad del sistema (arquitectura).

## A.4. Especificación de interfaces

Como se comentó en la sección A.3, cuando se captura un algoritmo en Vivado HLS, los argumentos de la *top function* actúan como los puertos del diseño hardware.

Cada uno de los argumentos, se comportará de forma diferente en función de su tipo y del uso que se

haga de él en el interior del diseño.

La elección de la interfaz es el primer paso que ha de darse a la hora de abordar un diseño, puesto que después será la forma de comunicarlo con los módulos de mayor jerarquía en los que se integre.

Por defecto, Vivado HLS sigue una serie de reglas a la hora de elegir que puertos crear a partir de los argumentos de la *top function*. De forma resumida:

- Si un argumento es leído desde el interior de la función pero nunca modificado, se implementará como una entrada.
- Si un argumento es leído y modificado, se corresponderá como un puerto de entrada/salida.
- Si un argumento es modificado pero no leído, se comportará como una salida.
- Los argumentos pasados por valor no pueden utilizarse para sacar datos al exterior.
- Los array se sintetizan como interfaces con memoria externa (ROM, RAM, FIFO,...)
- De forma predeterminada se generan señales de *handshaking* para controlar el diseño.

Como se verá a continuación, el comportamiento predeterminado puede modificarse mediante directivas.

#### A.4.1. Variables escalares

Para ejemplificar el comportamiento de una función cuyos argumentos son escalares, se presenta la siguiente *top function* disponible en el proyecto de ejemplo *sum\_io* de la versión 2013.4 de Vivado HLS.

```

1  #include "sum_io.h"

3  dout_t sum_io(din_t in1, din_t in2, dio_t *sum) {

5      dout_t temp;

7      *sum = in1 + in2 + *sum;
      temp = in1 + in2;

9      return temp;
11 }
```

Cuando se realiza la síntesis de esta función sin dar ninguna directiva adicional, los puertos presentes en el diseño hardware final son los mostrados en la figura A.3a. De ella se puede extraer la siguiente información:

1. En rojo aparecen los puertos **ap\_[nombre]** que controlan el bloque, como no aparece una señal de reloj se deduce que el diseño es combinacional.
  - **ap\_start** es la señal de habilitación del diseño.
  - **ap\_done** indica que los datos de salida son válidos.
  - **ap\_idle** indica que el bloque no está funcionando.
  - **ap\_ready** indica que el bloque está listo para recibir nuevos datos en las entradas.
2. En azul encontramos el puerto donde se presenta el valor de retorno de la función, denominado **ap\_return**.
3. En verde encontramos las entradas generadas por los pasos por valor de **in1** e **in2**

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	sum_io	return value
ap_rst	in	1	ap_ctrl_hs	sum_io	return value
ap_start	in	1	ap_ctrl_hs	sum_io	return value
ap_done	out	1	ap_ctrl_hs	sum_io	return value
ap_idle	out	1	ap_ctrl_hs	sum_io	return value
ap_ready	out	1	ap_ctrl_hs	sum_io	return value
ap_return	out	32	ap_ctrl_hs	sum_io	return value
in1	in	32	ap_hs	in1	scalar
in1_ap_vld	in	1	ap_hs	in1	scalar
in1_ap_ack	out	1	ap_hs	in1	scalar
in2	in	32	ap_vld	in2	scalar
in2_ap_vld	in	1	ap_vld	in2	scalar
sum_i	in	32	ap_hs	sum	pointer
sum_i_ap_vld	in	1	ap_hs	sum	pointer
sum_i_ap_ack	out	1	ap_hs	sum	pointer
sum_o	out	32	ap_hs	sum	pointer
sum_o_ap_vld	out	1	ap_hs	sum	pointer
sum_o_ap_ack	in	1	ap_hs	sum	pointer

(a) Sin directivas

(b) Con directivas

Figura A.3: Puertos generados en la síntesis de la función `sum_io`

- En naranja se han creado sendos puertos de entrada y salida correspondientes al paso por referencia de `sum` puesto que se realizan operaciones de lectura y escritura sobre el, además por defecto se añade una señal que indica que la información en `sum_o` es válida.

Para modificar la interfaz predeterminada, se utilizan directivas que pueden ser situadas o bien en el código fuente, mediante directivas `#pragma` de preprocesador, o bien en el script `directives.tcl`. En ambos casos es posible utilizar una interfaz gráfica para introducirlas.

La segunda opción es preferible frente a la inclusión directa en el código fuente, puesto que permite generar diferentes versiones del hardware a partir de un mismo código fuente, al existir un script `tcl` asociado a cada solución del proyecto.

Por ejemplo, si se desea utilizar un protocolo de *handshaking* junto con los parámetros `in1` y `sum` de la función, y una interfaz de dato válido con el parámetro `in2`, el contenido del script `directives.tcl` deberá ser el siguiente.

```
set_directive_interface -mode ap_hs "sum_io" in1
set_directive_interface -mode ap_vld "sum_io" in2
set_directive_interface -mode ap_hs "sum_io" sum
```

En la figura A.3b puede observarse el nuevo aspecto del esquema de entrada/salida del diseño.

- En `in1` e `sum` podemos comprobar que se dispone de señales de `ap_vld` y `ap_ack`.
  - Cuando el puerto con la interfaz de *handshaking* es de entrada, la primera se comporta como una señal de *load* indicando al bloque que tiene un dato de entrada disponible, siendo `ap_ack` la señal encargada de indicar que el dato ha sido capturado.
  - Cuando el puerto es de salida `ap_vld` nos indicará que el dato que se encuentra en el puerto es válido y `ap_ack` deberá ser activada en este momento para que el bloque pueda seguir funcionando.
- En `in2` encontramos simplemente una señal de `ap_vld` para indicar al bloque que hay nuevos datos en `in2`.

Como comentario adicional, puede observarse que ahora se ha añadido una señal de reloj para acompañar a los protocolos de *handshaking*. Que el hardware generado sea combinacional o secuencial,

depende de diferentes factores, como los protocolos utilizados, la complejidad de la arquitectura y la frecuencia de reloj que se marque como objetivo.

### A.4.2. Arrays y estructuras

Los arrays son implementados como interfaces con memoria externa al diseño. Por ejemplo

- Si se realizan lecturas y escrituras en posiciones no consecutivas sobre un array, se generará una interfaz estándar con una memoria (bus de datos, bus de direcciones y señal de lectura/escritura).
- Si el acceso es secuencial, puede generarse una interfaz para interactuar con una memoria FIFO.

Las optimizaciones que pueden llevarse a cabo sobre la interfaz generada a partir de los arrays, suelen ir destinadas a incrementar la cantidad de datos que el bloque puede recibir en cada ciclo de reloj, para evitar cuellos de botella. Este tema se tratará con más profundidad en la sección A.5.

El único aspecto destacable de las estructuras es que, de cara a especificar una interfaz, sus miembros se comportan de manera independiente. Es decir, si se dispone de una estructura con un array y un escalar, cuando se genere la interfaz se generará de forma independiente para el escalar y para el array, sin afectar al diseño que estén englobados dentro de una estructura.

### A.4.3. Interfaz AXI

Además del uso de señales de control, también es posible especificar directamente el uso de un bus estándar para conectar el módulo diseñado directamente a un procesador. El bus ofrecido es el AXI4 en sus variantes (Lite, Stream y Master), parte del estándar AMBA de ARM.

La ventaja del uso del bus AXI para incluir el módulo hardware generado en un diseño con procesador, es la posibilidad de generar automáticamente drivers para controlarlo.

### A.4.4. Especificación manual de interfaces

Finalmente, si las posibilidades ofrecidas no se adecúan a la interfaz necesaria para interactuar con el resto del diseño, es posible especificar manualmente el comportamiento de los puertos de entrada/salida. Para ello se proporcionan dos mecanismos:

- Bloques de código en los que se garantiza que no se insertarán ciclos de reloj de forma automática entre las instrucciones, y se desactivan las optimizaciones del compilador.
- Inserción manual de ciclos  $d < e$  reloj.

Si se opta por esta última posibilidad, las simulaciones del diseño una vez convertido a hardware deberán ser realizadas por el usuario, no pudiéndose generar el test bench de forma automática.

## A.5. Descripción de la funcionalidad y optimización

En esta sección se discuten las posibilidades existentes, de cara a especificar el comportamiento interno de los diseños.

Dentro de los elementos del lenguaje C, los más destacables a la hora de codificar la *top function* y que más afectan al rendimiento del hardware generado son:

- Tipos de datos
- Operadores

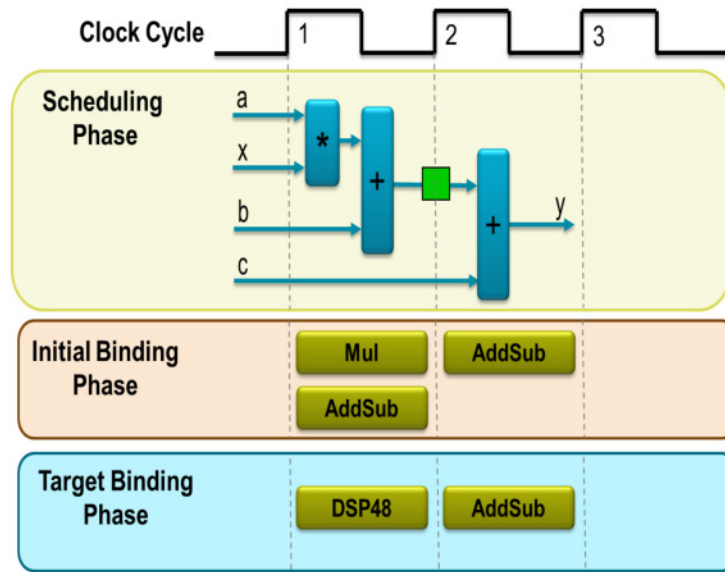


Figura A.4: Fases en el proceso de síntesis realizado por Vivado HLS

- Bucles
- Arrays

En primer lugar, se describirá el proceso de conversión que se lleva a cabo para transformar la *top function*, en una descripción RTL de funcionalidad equivalente. En el resto de apartados, se tratarán los elementos mencionados anteriormente, comentando su función dentro de los diseños y que opciones hay para optimizar su comportamiento.

### A.5.1. Proceso de conversión

El trabajo llevado a cabo por el sintetizador de Vivado HLS, puede dividirse en dos tareas principales:

- Un proceso de planificación (*scheduling*), en el que se decide cuantos ciclos de reloj son necesarios para ejecutar la función y que operaciones se han de llevar a cabo en cada uno.
- Un proceso en el que se asocian los operadores a recursos hardware concretos (sumadores, multiplicadores, unidades de coma flotante, memorias, ...), denominado *binding*.

En la figura A.4, extraída del manual de usuario proporcionado por Xilinx [1], pueden observarse de forma esquemática ambos procesos, cuando se genera el hardware equivalente a la siguiente función.

```

1  int foo(char x, char a, char b, char c){
    char y;
3   y = x*a+b+c;
    return y;
5  }
```

En primer lugar, se determina que en el primer ciclo de reloj puede realizarse la lectura de los datos y la multiplicación con suma, realizando la segunda suma en un segundo ciclo de reloj y escribiendo el resultado en la salida.

En la fase de asignación de recursos, primero se analiza la secuencia de operaciones y después, en función de la FPGA seleccionada, se asocian a los recursos más adecuados. Por ejemplo, la suma con

acumulación es asignada a un bloque DSP48E, que incluye un multiplicador acumulador.

Al igual que ocurría en el proceso de especificación de interfaces, Vivado HLS tiene un comportamiento predeterminado, que puede ser modificado mediante directivas. A continuación se presentan los puntos fundamentales:

- Como se comentó en la sección A.4, los argumentos de la *top function* se implementan como los puertos de entrada/salida del diseño RTL.
- La jerarquía de las funciones C se mantiene en el diseño RTL, a no ser que el compilador realice optimizaciones decidiendo convertirlas en funciones *inline*.
- Por defecto los bucles comparten los recursos hardware, implementándose como máquinas de estados.
- Los array se sintetizan como memorias RAM/ROM dependiendo de como se utilicen.

En los apartados siguientes, se comentan varias formas de influir sobre este comportamiento predeterminado, para adaptar las características del diseño a las especificaciones de partida.

### A.5.2. Tipos de datos

Todos los tipos de datos estándar de C, son soportados por el compilador de Vivado HLS. Además, por razones de optimización, hay disponible una librería de tipos de datos, que permite especificar variables de precisión arbitraria. El archivo de cabecera de la librería en el lenguaje C es `ap_cint.h`.

Para declarar variables de este tipo, se utiliza la sintaxis siguiente:

- `int[W]`
- `uint[W]`

Siendo [W] el ancho de palabra deseado. Gracias a estos tipos de datos se evita el consumo de recursos extra que se produciría al ceñirse a los tipos de datos estándar de 8, 16, 32 y 64 bits.

El principal inconveniente que se presenta es que, en el caso del lenguaje C, no es posible utilizar el depurador para analizar el diseño, siendo necesario utilizar alternativas como `printf` o sentencias `assert`, para llevar a cabo las tareas de depuración y validación iniciales.

### A.5.3. Operadores

Todos los operadores del lenguaje C estándar están soportados, dependiendo el hardware generado a partir de cada uno de varios factores:

- El tipo de datos utilizado.
- El dispositivo sobre el que deba implementarse el hardware.
- Las directivas dadas por el diseñador.

La mayor influencia del tipo de datos en el hardware se da cuando se trabaja con tipos de datos enteros o tipos de datos en coma flotante. Las operaciones en coma flotante se asocian a *cores* de coma flotante mientras que en el caso de trabajar con números enteros pueden darse diferentes situaciones, como se vio en el proceso de conversión.

La influencia del dispositivo viene determinada principalmente por el número de recursos disponibles de un determinado elemento, por ejemplo los DSP48E, haciendo que en caso de terminarse sea necesario utilizar otros medios para implementar las operaciones como LUTs.

Sobre este último punto, el diseñador puede forzar que un operador sea implementado utilizando un determinado recurso de la FPGA utilizada. Por ejemplo, podría indicarse que un sumador sea implementado sobre un bloque DSP48E.

Por otro lado, también es posible limitar el número de instancias hardware de un determinado operador, mediante la directiva `set_directive_allocation`. Por ejemplo, si se quisiera limitar el número bloques hardware asociados a la suma a 10, se procedería como sigue:

```
set_directive_allocation [OPTIONS] <location> <instances>
set_directive_allocation -limit <location> add
```

Siendo `<location>` el entorno en el que se quiere aplicar la limitación. En el caso de ser aplicable a todo el código, se sustituiría por el nombre de la *top function*.

Otro aspecto relativo a la optimización de los operadores que conviene comentar, es la asociación. Por defecto, el sintetizador trata de reducir la latencia combinando todas las operaciones posibles dentro de un mismo ciclo de reloj. Esta fase de la síntesis se denomina *expression balance* y puede ser desactivada en una determinada zona del diseño mediante la directiva

```
set_directive_expression_balance -off <location>
```

#### A.5.4. Bucles

Los bucles son la parte central de la mayoría de los diseños realizados en Vivado HLS. Esto es debido a que permiten capturar de forma sencilla algoritmos iterativos, en una zona limitada de código, siendo posible aplicar diferentes optimizaciones sobre ellos.

Para explicar los tipos de bucles y las diferentes optimizaciones que se pueden realizar sobre ellos, se va a utilizar como apoyo uno de los ejemplos del tutorial de Vivado HLS proporcionado por Xilinx [18].

La siguiente función escrita en lenguaje C, implementa de forma directa la multiplicación de dos matrices cuadradas  $A, B \in \mathbb{R}^{3 \times 3}$  utilizando bucles anidados.

```
1 #include "matrixmul.h"
3 void matrixmul(
    mat_a_t a[MAT_A_ROWS][MAT_A_COLS],
5    mat_b_t b[MAT_B_ROWS][MAT_B_COLS],
    result_t res[MAT_A_ROWS][MAT_B_COLS])
7 {
    // Iterate over the rows of the A matrix
9    Row: for(int i = 0; i < MAT_A_ROWS; i++) {
        // Iterate over the columns of the B matrix
11    Col: for(int j = 0; j < MAT_B_COLS; j++) {
        res[i][j] = 0;
13        // Do the inner product of a row of A and col of B
        Product: for(int k = 0; k < MAT_B_ROWS; k++) {
15            res[i][j] += a[i][k] * b[k][j];
        }
17    }
    }
19 }
```

Siendo los tipos de datos y constantes utilizadas

```
1 #define MAT_A_ROWS 3
   #define MAT_A_COLS 3
```



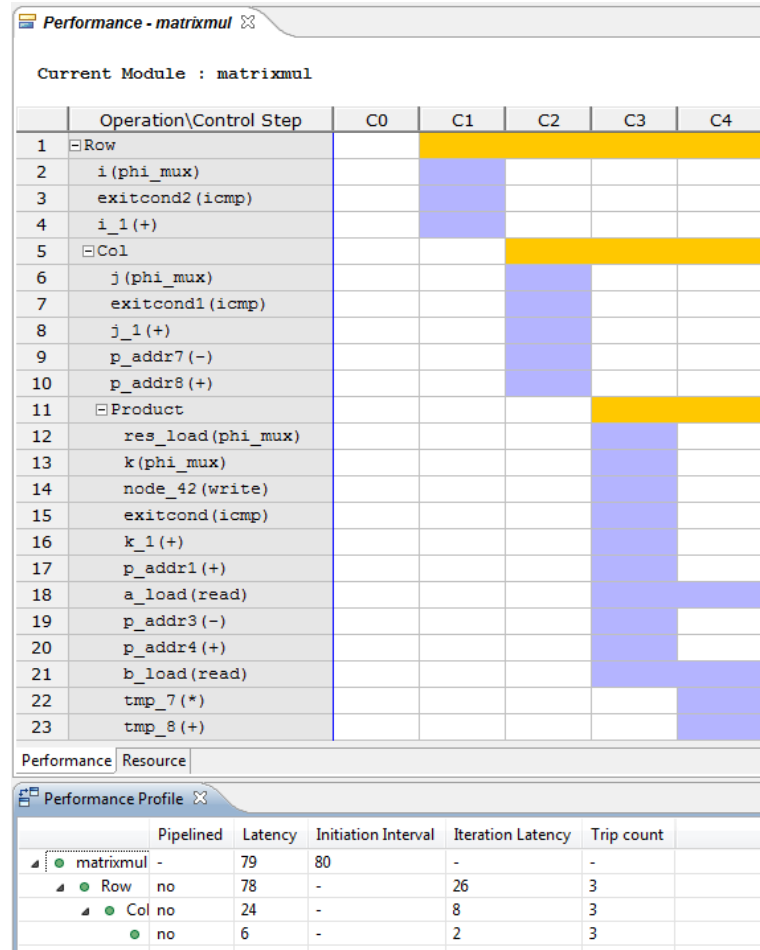


Figura A.5: Análisis del multiplicador de matrices sin optimizar

```

3 #define MAT_B_ROWS 3
  #define MAT_B_COLS 3
5
  typedef char mat_a_t;
7 typedef char mat_b_t;
  typedef short result_t;

```

En el código fuente de la función `matrixmul`, pueden observarse tres bucles anidados con las etiquetas: `Row`, `Col` y `Product`. Los bucles `Row` y `Col` recorren los nueve elementos de la matriz de salida `res`, calculando el resultado mediante la expresión (A.1) en el bucle `Product`.

$$res_{ij} = \sum_{k=0}^{k=2} a_{ij} \cdot b_{ij} \quad (A.1)$$

Cuando se sintetiza la función mostrada sin ningún tipo de optimización, se genera una máquina de estados que gestiona las operaciones y la lógica asociada a las lecturas y escrituras de datos. Como ya se comentó, por defecto los bucles utilizan los mismos recursos hardware para realizar todas las iteraciones.

En la figura A.5, puede verse el análisis funcional del hardware generado. Las operaciones han sido organizadas en una máquina de 5 estados, donde C0 es el estado de reposo y los cuatro restantes

desarrollan el cálculo de los elementos de la matriz `res`.

Lo primero que puede observarse es que pasar de un bucle a otro, requiere un ciclo de reloj, puesto que implica una transición en la máquina de estados. En la parte inferior del diagrama, se muestra el número de ciclos de reloj consumidos por cada bucle. Estos resultados pueden justificarse a partir del diagrama como sigue.

El bucle interior (`Product`) necesita dos ciclos de reloj para realizar una iteración, luego su latencia vendrá dada por la expresión

$$T_{Product} = 3 \cdot T_{it,Product} = 3 \cdot 2 \cdot T_{CLK} = 6 \cdot T_{CLK} \quad (A.2)$$

Siendo 3 el número de iteraciones.

Si nos centramos ahora en el bucle `Col`, el tiempo que tarda en realizar una iteración vendrá determinado por el tiempo que tarda en ejecutarse el bucle interior, y la entrada y salida al mismo. Es decir:

$$T_{Col} = 2 \cdot T_{CLK} + T_{Product} = 8 \cdot T_{CLK} \quad (A.3)$$

La latencia del bucle exterior `Row` se obtiene de forma análoga, llegando a:

$$T_{Row} = 78 \cdot T_{CLK} \quad (A.4)$$

Finalmente, el diseño total requiere un ciclo de reloj extra para finalizar la ejecución (vuelta al estado `C0`), y un ciclo de reloj más para empezar a procesar un nuevo set de datos, determinándose así la latencia y el intervalo.

Puesto que entre los bucles `Row` y `Col` no hay ninguna operación, en el ciclo de reloj empleado al saltar de uno a otro no ocurre nada luego ambos podrían ser combinados en un único bucle.

Para hacer esto puede, o bien escribirse código fuente en consecuencia, o bien utilizarse la directiva de optimización:

```
set_directive_loop_flatten <location>
```

que combinará automáticamente los bucles al generar el hardware en un nuevo bucle llamado `Row_Col`. En la figura A.6a puede verse que la nueva máquina de estados tiene un estado menos que la anterior, reduciéndose la latencia del diseño a  $73 \cdot T_{CLK}$ .

A pesar de que así se reduce la latencia, es obvio que no hay ningún tipo de paralelismo en el hardware. La opción más inmediata, es paralelizar todas las operaciones del bucle más interior (`Product`).

En la figura A.6b se muestra la máquina de estados generada, tras aplicar la directiva

```
set_directive_unroll "matrixmul/Product"
```

sobre el bucle interno. Cuando se aplica una operación de *unroll* sobre un bucle, este desaparece del diseño, generándose un hardware equivalente que realiza todas las operaciones de la forma más concurrente posible. La clave a la hora de utilizar esta técnica, es estudiar las dependencias de datos entre las iteraciones del bucle sobre el que se utiliza.

Ahora la latencia del diseño es de  $37 \cdot T_{CLK}$ . A pesar de haberse conseguido una mejora sustancial respecto a la implementación anterior, si se observa la secuencia de operaciones sobre la memoria, estas no se han paralelizado el máximo posible.

En diseño digital, una de las técnicas más utilizadas para mejorar la velocidad de un diseño, sin incrementar de forma excesiva el consumo de recursos, es la segmentación. Es decir, empezar a procesar nuevos datos cuando aun no se ha terminado de operar con los anteriores.

Si aplicamos segmentación al bucle intermedio, el diseño final se verá afectado en diferentes aspectos:

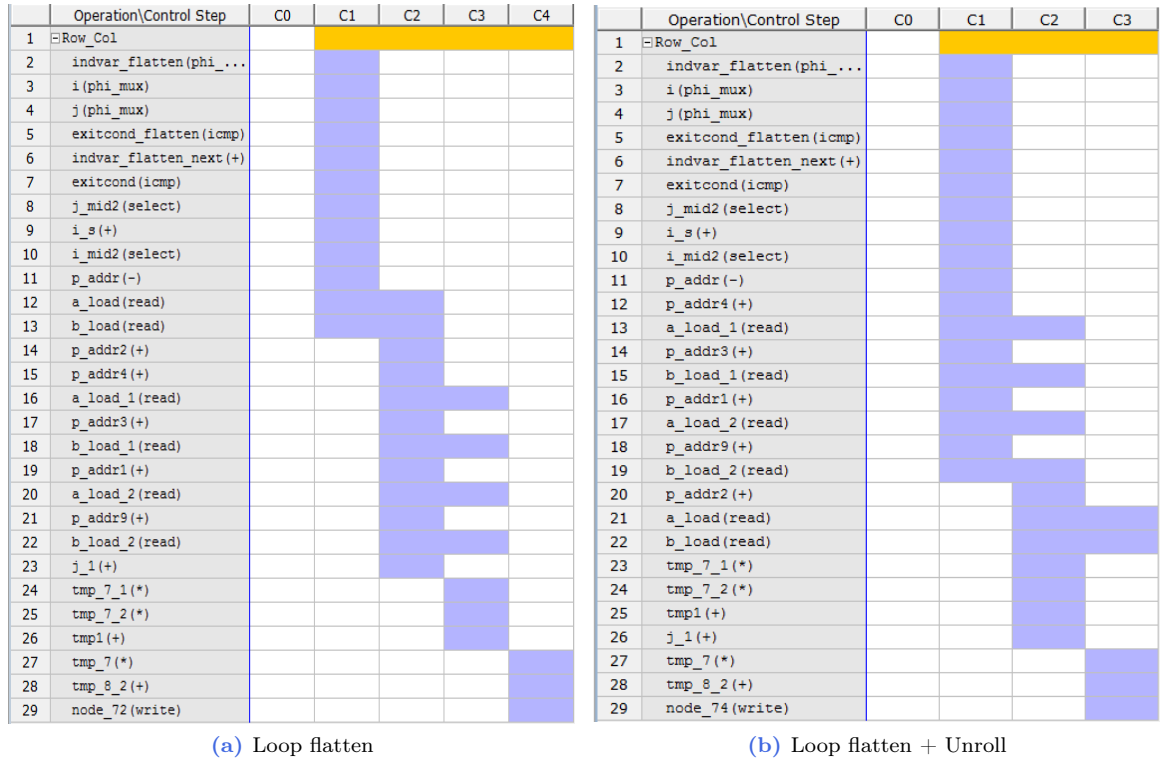


Figura A.6: Análisis del multiplicador de matrices tras varias optimizaciones

- El bucle interior sufrirá un *unroll*.
- Los bucles Row y Col serán combinados (*flatten*).
- El hardware generado a partir de las operaciones situadas en el bucle interior, será capaz de realizar las lecturas y escrituras en memoria utilizando el máximo ancho de banda posible.

Para realizar la operación de segmentación sobre el bucle Col se procede como sigue:

```
set_directive_pipeline "matrixmul/Col"
```

Por defecto, cuando se realiza un *pipeline* sobre una función o un bucle, Vivado HLS intenta conseguir el menor intervalo de iniciación posible (*Initiation Interval*). Es decir, en el mejor de los casos el sistema generará un resultado por ciclo de reloj, transcurrida una latencia inicial.

Tras esta optimización, la latencia del diseño pasa a ser de  $20 \cdot T_{CLK}$ . Para ver en que punto se ha

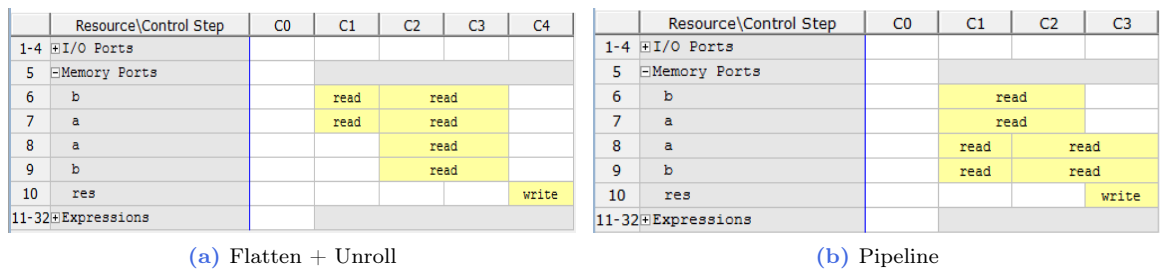


Figura A.7: Accesos a memoria del multiplicador de matrices tras diferentes optimizaciones

producido la mejora respecto a la opción anterior, en la figura A.7 se muestran las operaciones sobre las memorias asociadas a los puertos de entrada/salida.

Se puede observar que, en la solución que utiliza *Pipeline* en el bucle *Col*, las operaciones de lectura y escritura están solapadas, eliminándose el último estado en el que se únicamente se escribía el resultado.

Otro punto destacable de esta implementación, es el intervalo de iniciación conseguido. En el informe generado tras la síntesis, se indica que el  $II^3$  conseguido es de 2, siendo el objetivo  $II = 1$ . En otras palabras, el bucle genera un resultado nuevo cada dos ciclos de reloj, transcurrido el periodo de latencia.

El motivo por el que no se ha alcanzado el intervalo objetivo, puede averiguarse observando los avisos producidos en la implementación.

```
@W [SCHED-69] Unable to schedule 'load' operation ('a_load', matrixmul.cpp:16) on
array 'a' due to limited memory ports.
```

Llegados a este punto, si se quiere continuar mejorando el diseño actuando sobre el cuerpo de la *top function*, es necesario trabajar sobre los accesos a memoria, puesto que se han convertido en el factor limitante.

### A.5.5. Arrays

De forma similar a lo que ocurre en la especificación de interfaces, los arrays son implementados como memorias. Por defecto, el tipo de memoria estará determinado por el código fuente relacionado con la variable.

De forma predeterminada, Vivado HLS asigna memorias RAM a los array. Como se vio en el apartado anterior, conforme se optimiza un diseño aparecen cuellos de botella si hay memorias involucradas en la operación del algoritmo.

Para incrementar el ancho de banda del sistema, existen diferentes posibilidades.

- Memorias Dual Port.
- Utilizar memorias con un ancho de palabra mayor, transmitiendo varios datos en paralelo (*array reshape*).
- Utilizar varias memorias o una memoria con un bus de datos más ancho, reordenando su contenido (*array partition/reshape*).

Las opciones anteriores son válidas de cara a los array utilizados en la interfaz y los array utilizados de forma interna.

Si el ancho de banda conseguido tras aplicar estas optimizaciones no es suficiente, es posible implementar los array internos en forma de registros, si los recursos consumidos no son un problema.

Retomando el ejemplo del multiplicador de matrices, en la figura A.8 se muestra la interfaz generada en la solución donde se segmenta el bucle *Col*. Si se observan los puertos correspondientes a los argumentos *a*, *b* y *res* de la *top function* puede comprobarse que, aunque no se ha especificado de forma explícita, el sintetizador los ha implementado como interfaces con memorias DP.

Salvo que se especifique mediante directivas, este es el máximo esfuerzo que Vivado HLS realiza para aumentar el ancho de banda. Ahora se abren dos posibilidades: aplicar optimizaciones a la función completa, para intentar paralelizar no solo las operaciones sobre el cálculo de un elemento, sino las del cálculo del resultado completo; o intentar resolver el problema de acceso a memoria.

---

<sup>3</sup>Initiation Interval

Interface					
Summary					
RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	matrixmul	return value
ap_rst	in	1	ap_ctrl_hs	matrixmul	return value
ap_start	in	1	ap_ctrl_hs	matrixmul	return value
ap_done	out	1	ap_ctrl_hs	matrixmul	return value
ap_idle	out	1	ap_ctrl_hs	matrixmul	return value
ap_ready	out	1	ap_ctrl_hs	matrixmul	return value
a_address0	out	4	ap_memory	a	array
a_ce0	out	1	ap_memory	a	array
a_q0	in	8	ap_memory	a	array
a_address1	out	4	ap_memory	a	array
a_ce1	out	1	ap_memory	a	array
a_q1	in	8	ap_memory	a	array
b_address0	out	4	ap_memory	b	array
b_ce0	out	1	ap_memory	b	array
b_q0	in	8	ap_memory	b	array
b_address1	out	4	ap_memory	b	array
b_ce1	out	1	ap_memory	b	array
b_q1	in	8	ap_memory	b	array
res_address0	out	4	ap_memory	res	array
res_ce0	out	1	ap_memory	res	array
res_we0	out	1	ap_memory	res	array
res_d0	out	16	ap_memory	res	array

Figura A.8: Interfaz del multiplicador de matrices segmentado

En primer lugar se va a optar por la segunda opción. Para resolver el inconveniente de los puertos insuficientes en la memoria correspondiente al parámetro *a*, se puede reorganizar la forma utilizada para interactuar con el exterior, utilizando varias memorias con los datos intercalados, o una memoria con bus de datos mayor que sea capaz de proporcionar múltiples datos en cada lectura.

Desde el punto de vista ideal, ambas opciones llevarán al mismo resultado, luego para decidir hay que estudiar los recursos disponibles. En este caso se realizará una optimización del tipo *array reshape*, es decir, memorias un un bus de datos más ancho. Esta directiva de optimización tiene la sintaxis siguiente:

```
set_directive_array_reshape [OPTIONS] <location> <array>
```

Antes de continuar, es conveniente comentar algunas opciones disponibles: `-dim <integer>`, `-factor <integer>` y `-type (block|cyclic|complete)`. La primera hace referencia a que dimensión del array se desea reorganizar, si no se especifica la optimización se aplicará a todas las dimensiones del array.

La opción `-factor` hace referencia al número de valores que serán colocados en una posición de memoria del nuevo array. Es decir, el ancho de palabra del nuevo bus de datos ( $WL'$ ), vendrá dado por la expresión:

$$WL' = N \cdot WL \quad (A.5)$$

Donde  $WL$  es el ancho de palabra de los datos antes de la reorganización y  $N$  se corresponde con el valor de la opción `-factor`.

Finalmente, las opciones `block` y `cyclic` de `-type`, crean bloques iguales, consecutivos o alternativos respectivamente, mientras que `complete` une todos los datos de una dimensión en un único elemento.

Si se atiende a la línea de código fuente que implementa el cálculo de los elementos de la matriz resultado:

```
res[i][j] += a[i][k] * b[k][j];
```

puede observarse que en cada ejecución se mantiene fija una de las dimensiones, iterando sobre la otra. A la vista de esto, se puede determinar que en la variable *a* se deberá reorganizar la dimensión 2 y,

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	matrixmul	return value
ap_rst	in	1	ap_ctrl_hs	matrixmul	return value
ap_start	in	1	ap_ctrl_hs	matrixmul	return value
ap_done	out	1	ap_ctrl_hs	matrixmul	return value
ap_idle	out	1	ap_ctrl_hs	matrixmul	return value
ap_ready	out	1	ap_ctrl_hs	matrixmul	return value
a_address0	out	2	ap_memory	a	array
a_ce0	out	1	ap_memory	a	array
a_q0	in	24	ap_memory	a	array
b_address0	out	2	ap_memory	b	array
b_ce0	out	1	ap_memory	b	array
b_q0	in	24	ap_memory	b	array
res_address0	out	4	ap_memory	res	array
res_ce0	out	1	ap_memory	res	array
res_we0	out	1	ap_memory	res	array
res_d0	out	16	ap_memory	res	array

(a) Interfaz

Resource\Control Step	C0	C1	C2	C3
1-4 I/O Ports				
5 Memory Ports				
6 a		read		
7 b		read		
8 res				write
9 Expressions				
10 exitcond_flatten_fu...		icmp		
11 exitcond_fu_142		icmp		
12 j_mid2_fu_148		select		
13 i_mid2_fu_162		select		
14 j_1_fu_180		+		
15 i_s_fu_156		+		
16 indvar_flatten_next...		+		
17 i_phi_fu_111		phi_mux		
18 indvar_flatten_phi...		phi_mux		
19 j_phi_fu_122		phi_mux		
20 tmp_7_2_fu_256			*	
21 tmp_7_1_fu_222			*	
22 tmp1_fu_262			+	
23 p_addr_fu_285				-
24 tmp_7_fu_316				*
25 p_addr1_fu_295				+
26 tmp_8_2_fu_322				+

(b) Actividad de los recursos

**Figura A.9:** Implementación del multiplicador de matrices con Pipeline + Reshape

en la variable **b**, la dimensión 1. Además, se hará de forma completa, resultando esto en una interfaz con dos memorias que almacenan 3 datos de  $3 \times 8 = 24\text{bits}$ .

	Latencia	Intervalo	Iteraciones
Bucle	10	1	3
Diseño	12	13	-

**Tabla A.1:** Latencia del bucle y del diseño en el multiplicador de matrices con Pipeline + Reshape

Las directivas que se añadirán para conseguir este efecto son:

```
set_directive_pipeline "matrixmul/Col"
set_directive_array_reshape -type complete -dim 2 "matrixmul" a
set_directive_array_reshape -type complete -dim 1 "matrixmul" b
```

En la figura A.9, se muestran los nuevos accesos a memoria, la interfaz generada y la latencia del bucle tras la reorganización de los datos.

En la interfaz puede observarse que, ahora, el bus de datos correspondiente a las variables **a** y **b** tiene 24 bits de ancho y el bus de direcciones se compone de 2 bits. Atendiendo a las operaciones de lectura/escritura de la nueva máquina de estados, puede verse que ahora solo es necesario un acceso a memoria en cada iteración.

Además, se ha conseguido segmentar por completo el hardware que implementa el bucle, con un  $II = 1$  y una latencia de  $T_{RowCol} = 3 \cdot T_{CLK}$  (A.1). A partir de las características del bucle, se justifica la latencia del diseño como:

$$T_L = T_{CLK} + T_{RowCol} + (N - 1) \cdot T_{CLK} \quad (\text{A.6})$$

siendo  $N$  el número de iteraciones que realiza el bucle **Row\_Col** y correspondiéndose el último  $T_{CLK}$

al paso del estado de reposo al primer estado de la **fsm!**<sup>4</sup>.

Por otro lado, se necesita otro ciclo de reloj para comenzar a procesar nuevos datos, puesto que se ha de volver al estado de reposo al terminar la operación. Esto indica que, a pesar de haber segmentación en el hardware correspondiente al bucle, el diseño no está segmentado.

### A.5.6. Optimización de la función completa

En las secciones anteriores, se ha visto como pueden mejorarse las características de un diseño aplicando diferentes optimizaciones sobre los elementos que lo componen. También es posible actuar sobre el diseño de forma global, aplicando directivas de optimización directamente sobre la *top function*.

Las posibilidades son más reducidas, pero es sencillo conseguir buenos resultados en cuanto a tiempo de ejecución. El principal inconveniente de las optimizaciones globales, es que generalmente, implican un gran aumento en el número de recursos consumidos.

En primer lugar, volviendo sobre el ejemplo del multiplicador de matrices, se va a mostrar el efecto de segmentar el diseño completo, en lugar del bucle interior. Para ello, se utiliza la misma directiva, pero seleccionando la *top function* como localización.

```
set_directive_pipeline "matrixmul"
```

En la tabla A.2 se muestra una comparación, en términos de latencia y recursos consumidos, de las diferentes alternativas de optimización planteadas.

Ahora, el hardware generado es capaz de procesar un par de matrices cada cinco ciclos de reloj, tras una latencia inicial. También puede verse que el consumo de recursos, sobretudo el de módulos DSP48E, se ha incrementado. Esto es debido a que, al aplicar segmentación a una zona del código, todos los bucles interiores se deshacen (*unroll*).

	Pipeline Col	Pipeline Col + Reshape	Pipeline Top	Sin optimizar
BRAM_18K	0	0	0	0
DSP48E	3	3	27	1
FF	56	55	485	43
LUT	70	37	64	62
Latencia ( $T_{CLK}$ )	20	12	5	79
Intervalo ( $T_{CLK}$ )	21	13	8	80

**Tabla A.2:** Latencia de las diferentes alternativas de optimización y consumo de recursos

Para terminar, se va a comentar otra posibilidad a la hora de optimizar el diseño aplicando directivas globales, que permite conseguir segmentación sin recurrir a deshacer los bucles completamente.

Si dentro de un diseño se dispone de varios bucles consecutivos, y uno depende de los datos generados por el otro, el hardware generado por defecto no contemplará la posibilidad de mandar los datos al segundo bucle según estén disponibles, sino que se implementarán de forma secuencial.

Una forma de solucionar este problema, es el uso de la directiva **dataflow**. Para ejemplificar este comportamiento, se ha extendido el multiplicador de matrices, realizándose ahora la multiplicación de tres matrices ( $A, B, C \in \mathbb{R}^{3 \times 3}$ ). A continuación se muestra el código fuente de la nueva *top function*.

```
#include "matrixmul.h"
2
void matrixmul2(
4     char a[MAT_A_ROWS][MAT_A_COLS],
```

<sup>4</sup>fsm!

```

    char b[MAT_B_ROWS][MAT_B_COLS],
6    char c[MAT_C_ROWS][MAT_C_COLS],
    int res[MAT_A_ROWS][MAT_C_COLS])
8 {
    short res_tmp[MAT_A_ROWS][MAT_B_COLS];

10
    Row1: for(int i = 0; i < MAT_A_ROWS; i++) {
12        Col1: for(int j = 0; j < MAT_B_COLS; j++) {
            res_tmp[i][j] = 0;
14            Product1: for(int k = 0; k < MAT_B_ROWS; k++) {
                res_tmp[i][j] += a[i][k] * b[k][j];
16            }
        }
18    }

20    Row2: for(int i = 0; i < MAT_A_ROWS; i++) {
        Col2: for(int j = 0; j < MAT_B_COLS; j++) {
22            res[i][j] = 0;
            Product2: for(int k = 0; k < MAT_B_ROWS; k++) {
24                res[i][j] += res_tmp[i][k] * (short)c[k][j];
            }
26        }
    }
28 }

```

En la nueva función hay ahora seis bucles, agrupados tres a tres. El primer grupo realiza la operación  $R = A \cdot B$ , y el segundo  $R' = R \cdot C$ . Correspondiéndose  $R'$  y  $R$  con `res` y `res_tmp` respectivamente.

Para que los bucles Row1, Col1 y Product1, colaboren con Row2, Col2 y Product2, proporcionando resultados conforme son calculados, se aplica sobre la *top function* la directiva

```
set_directive_dataflow "matrixmul2"
```

Cuando se dispone de varios bucles relacionados dentro de una misma función, es posible combinar esta directiva con optimizaciones sobre el interior de la *top function*, para conseguir un diseño segmentado que no implique deshacer todos los bucles y, por tanto, un gran aumento en los recursos consumidos.

	Solution 1	Solution 2	Solution 3	Solution 4	Solution 5
BRAM	0	0	0	0	0
DSP48E	2	54	2	6	6
FF	132	1503	172	245	487
LUT	168	96	191	151	1386
Latencia	158	12	159	105	52
Intervalo	159	5	80	53	53

**Tabla A.3:** Consumo de recursos y latencia en el multiplicador de tres matrices, para diferentes alternativas de optimización

En la tabla A.3, se muestra el consumo de recursos y la latencia de la función `matrixmul2`, cuando se realizan las siguientes optimizaciones sobre ella

**Solution 1:** ninguna optimización.

**Solution 2:** *Pipeline* sobre la función principal.

**Solution 3:** *Dataflow* sobre la función principal.



**Solution 4:** *Dataflow* sobre la función principal y *unroll* sobre los bucles `Col1` y `Col2`.

**Solution 5:** Como la anterior, pero añadiendo *reshape* en `a`, `b`, `c` y `res_tmp`.

Si se observan los resultados, se puede comprobar como utilizando la directiva `dataflow`, puede llegarse a diseños segmentados. Por otro lado, como era de esperar, los mejores resultados siguen consiguiéndose cuando se aplica *pipeline* a la función completa de forma explícita.



## Apéndice B

# Manual de usuario

El objetivo de este apéndice es, por un lado, comentar en detalle como utilizar los diseños realizados (algoritmo CORDIC y método de Jacobi) como módulos independiente y, por otro lado, tratar brevemente los `script` Matlab utilizados para simular y analizar los algoritmos.

La versión de Vivado HLS utilizada en las últimas versiones es la incluida en *Vivado Suite 2014.1*, habiéndose comprobado su correcto funcionamiento también en la versión anterior (2013.4).

## B.1. Directorios del soporte informático

En el DVD que se incluye junto con el libro, encontramos el directorio **JacobiHLS**, donde se encuentran los proyectos de Vivado HLS y el código fuente tanto en C como de Matlab.

Dentro del directorio **JacobiHLS**, los elementos están distribuidos como sigue:

- El directorio **fuentes** contiene todos los archivos `*.c` correspondientes al algoritmo CORDIC y al método de Jacobi, además de un subdirectorio **ficheros** con los datos de prueba y los resultados de comprobación generados mediante MATLAB, almacenados en ficheros `*.dat`.
- En el directorio **matlab** se encuentran todos los ficheros `*.m` relacionados con la simulación, la generación de datos de test y el análisis del error.

## B.2. Generación del módulo CORDIC

Para generar el módulo CORDIC, en primer lugar se debe crear un proyecto en Vivado HLS. Para ello, puede utilizarse la interfaz gráfica siguiendo los pasos del asistente, o un script de comandos `tcl`.

Por comodidad, se ha incluido un script `cordic.tcl` que genera el proyecto con los ficheros de código fuente y test adecuados, bajo las condiciones expuestas durante el trabajo (señal de reloj de 100Mhz y dispositivo x).

El script debe ser ejecutado desde la consola de Vivado HLS (*Vivado HLS 2014.1 Command Prompt*). Los pasos para crear el proyecto a partir del script `cordic.tcl` son los siguientes:

- Abrir *Vivado HLS 2014.1 Command Prompt*.
- Situar en el directorio **JacobiHLS** (es necesario sacar la carpeta fuera del DVD) mediante el comando `cd <ruta>`.

- Ejecutar el script `cordic.tcl` mediante el comando `vivado_hls -f cordic.tcl`.
- Si todo va bien, encontraremos un nuevo directorio (`cordic_scp`) con el proyecto ya sintetizado para la arquitectura paralela.
- En el directorio `cordic_scp/PARALLEL/syn/vhdl` encontramos el código VHDL listo para utilizar.

A partir de este proyecto inicial pueden generarse nuevas soluciones (la arquitectura serie, por ejemplo) o exportarlo de forma directa a alguna otra herramienta de Xilinx. Para ello, puede utilizarse la interfaz gráfica, abriendo el proyecto desde la misma o mediante el comando:

```
vivado_hls -p cordic_scp
```

### B.3. Generación del módulo para el cálculo de autovalores y autovectores

De forma análoga al módulo CORDIC, se ha creado un script de comandos *tcl* para crear el proyecto en Vivado HLS, junto con la solución correspondiente a la arquitectura serie, en su versión con cuatro módulos CORDIC en modo rotación.

La manera de generar el proyecto y el código VHDL es similar:

- Abrir *Vivado HLS 2014.1 Command Prompt*.
- Situar en el directorio `JacobiHLS` (es necesario sacar la carpeta fuera del DVD) mediante el comando `cd <ruta>`.
- Ejecutar el script `jacobi.tcl` mediante el comando `vivado_hls -f jacobi.tcl`.
- Si todo va bien, encontraremos un nuevo directorio (`cordic_svd`) con el proyecto ya sintetizado para la arquitectura paralela.
- En el directorio `cordic_svd/Serial/syn/vhdl` encontramos el código VHDL listo para utilizar.
- Si se desea exportar el diseño a otra herramienta de Xilinx o explorar otras alternativas de implementación, puede abrirse el proyecto en la interfaz gráfica mediante el comando `vivado_hls -p cordic_svd`.

### B.4. Simulación del algoritmo CORDIC y el método de Jacobi en Matlab

Con objeto de comprobar la correcta funcionalidad de los algoritmos antes de realizar la implementación en C, se desarrollaron funciones equivalentes en Matlab que pueden encontrarse en el directorio `matlab`.

Por un lado, encontramos las funciones `cordic_scp.m` y `cordic_svd.m` que implementan la funcionalidad del algoritmo CORDIC y del método de Jacobi respectivamente.

Por otro lado, una serie de *script \*.m* que someten a prueba las funciones y representan de forma gráfica los resultados expuestos a lo largo de este trabajo.

# Código fuente de la implementación en Vivado HLS

## C.1. Módulo CORDIC

### C.1.1. cordic\_scp.c

```

1 #include "cordic_scp.h"
2 #include <ap_cint.h>

4 void
   cordic_scp( cordic_t xi, cordic_t yi,
6             cordic_t zi,
             cordic_t *xo, cordic_t *yo,
8             cordic_t *zo,
             uint1 mode){

10     // The final optimization directives are included as #pragma for the
12     // function to be used in the rest of the system.
   #pragma HLS PIPELINE II=1
14 #pragma HLS function_instantiate variable=mode

16     // Memory element containing micro-rotation angle values, it'll be a
   // ROM or a bank of registers, depending of the implementation.
18     cordic_t lut[MAX_ITERATIONS] = {25736, 15193, 8027, 4075, 2045,
                                       1024, 512, 256, 128, 64, 32,
20                                       16, 8, 4, 2, 1, 0};

22     cordic_t x = xi;
   cordic_t y = yi;
24     cordic_t z = zi;

26     // Input angle has to be registered in case a quadrant correction is
   // needed.
28     cordic_t ziRegIn = zi;

```

```

30 // Temporal variables to implement CORDIC hardware equations.
   cordic_t xyAux = 0;
32 cordic_t xTmp = 0;
   cordic_t yTmp = 0;
34
   // First quadrant correction.
36 if (mode == TRANSLATE){
   if ( (x < 0) && (y >= 0) ){
38     xyAux = y;
     y = -x;
40     x = xyAux;
     z -= FIXED_PI_2;
42   }
   else if ( (x < 0) && (y < 0) ){
44     xyAux = x;
     x = -y;
46     y = xyAux;
     z += FIXED_PI_2;
48   }
   }
50 else{ // Rotate.
   if (z > FIXED_PI_2){
52     z -= FIXED_PI_2;
   }
   else if( z < -FIXED_PI_2 ){
54     z += FIXED_PI_2;
56   }
   }
58
   // Loop implementing CORDIC hardware equations.
60 MAINLOOP: for (int i = 0; i < MAX_ITERATIONS; i++){
   if (mode == ROTATE){
62     xTmp = x;
     yTmp = y;
64     if (z < 0){
       z = z + lut[i];
66       x = xTmp + (yTmp >> i);
       y = yTmp - (xTmp >> i);
68     }
     else{
70       z = z - lut[i];
       x = xTmp - (yTmp >> i);
72       y = yTmp + (xTmp >> i);
     }
74   } // End of rotation mode.
   else{ // Translate.
76     xTmp = x;
     yTmp = y;
78     if (y > 0){
       z = z + lut[i];
80       x = xTmp + (yTmp >> i);
       y = yTmp - (xTmp >> i);
82     }
     else{

```

```

84         z = z - lut[i];
85         x = xTmp - (yTmp >> i);
86         y = yTmp +(xTmp >> i);
87     }
88 } // End of translation mode.
89 }
90
91 cordic_t xiRegOut = x;
92 cordic_t yiRegOut = y;
93 cordic_t xTmpOut = 0, yTmpOut = 0;
94
95 // Second quadrant correction.
96 if (mode == ROTATE){
97     if ( ziRegIn > FIXED_PI_2 ){
98         x = -yiRegOut;
99         y = xiRegOut;
100     }
101     else if ( ziRegIn < -FIXED_PI_2 ){
102         y = -xiRegOut;
103         x = yiRegOut;
104     }
105 }
106
107 // Scale factor compensation and data output.
108 // Data is converted to a wider data type to avoid
109 // truncation in the fixed point multiplication.
110 *xo = (cordic_t)( (((xy_mult_t)x)
111     * ((xy_mult_t)INV_K1)) >> XY_SCALE_FACTOR );
112
113 // Scaling factor doesn't need to be compensated in vectoring
114 // mode since y is forced to 0, so we can save a DSP48E when
115 // function_instantiate is used.
116 if (ROTATE == mode){
117     *yo = (cordic_t)( (((xy_mult_t)y)
118     * ((xy_mult_t)INV_K1)) >> XY_SCALE_FACTOR );
119 }
120 else{
121     *yo = (cordic_t)yiRegOut;
122 }
123 *zo = (cordic_t)z;
124 }

```

### C.1.2. cordic\_scp.h

```

1 #ifndef CORDIC_SCP
2 #define CORDIC_SCP
3
4 #include <ap_cint.h>
5
6 #define WORD_LENGTH 18
7 #define MAX_ITERATIONS 17
8
9 #define INPUT_SIZE 498
10

```

```

#define ROTATE 1
12 #define TRANSLATE 0

14 #define XY_SCALE_FACTOR 15
#define FIXED_PI_2 51472
16 #define INV_K1 19898

18 typedef int18 cordic_t;
typedef int18 xy_internal_t;
20 typedef int32 xy_mult_t;

22 void cordic_scp( cordic_t xi, cordic_t yi, cordic_t zi,
                  cordic_t *xo, cordic_t *yo, cordic_t *zo,
24                  const uint1 mode);

26 #endif

```

## C.2. Algoritmo de Jacobi

### C.2.1. cordic\_svd.c

```

#include "cordic_svd.h"
2
void cordic_svd(cordic_t COV_IN[MATRIX_SIZE][MATRIX_SIZE],
4             cordic_t EIGENVALUES[MATRIX_SIZE],
             cordic_t EIGENVECTORS[MATRIX_SIZE][MATRIX_SIZE])
6 {

8     // Loop variables.
    int h = 0;
10    int i = 0;
    int j = 0;

12
    cordic_t xi = 0, yi = 0, zi = 0;
14    cordic_t xo = 0, yo = 0, zo = 0;

16    cordic_t xo1 = 0, xo2 = 0, xo3, xo4;
    cordic_t yo1 = 0, yo2 = 0, yo3, yo4;
18    cordic_t zo1 = 0, zo2 = 0, zo3 = 0, zo4 = 0;

20    cordic_t Vz01 = 0, Vz02 = 0;

22    cordic_t rotation_angles[DIAGONAL_PROCESSORS];

24    // Auxiliar matrix to store intermediate results.
    cordic_t COV_IN_AUX[MATRIX_SIZE][MATRIX_SIZE];
26    cordic_t COV_IN_TMP[MATRIX_SIZE][MATRIX_SIZE];

28    cordic_t EIGENV_TMP[MATRIX_SIZE][MATRIX_SIZE];
    cordic_t EIGENV_AUX[MATRIX_SIZE][MATRIX_SIZE];
30

    // MxM identity matrix.
32    cordic_t EYE[MATRIX_SIZE][MATRIX_SIZE] = {{32768, 0, 0, 0, 0, 0, 0, 0, },

```



```

34     {0, 32768, 0, 0, 0, 0, 0, 0, },
35     {0, 0, 32768, 0, 0, 0, 0, 0, },
36     {0, 0, 0, 32768, 0, 0, 0, 0, },
37     {0, 0, 0, 0, 32768, 0, 0, 0, },
38     {0, 0, 0, 0, 0, 32768, 0, 0, },
39     {0, 0, 0, 0, 0, 0, 32768, 0, },
40     };

42     // Array direction wrapper to rearrange matrix elements
43     // in each iteration.
44     int dir[MATRIX_SIZE] = {0, 3, 1, 5, 2, 7, 4, 6};

46     // Input matrix storage in internal array to allow
47     // optimizations.
48     COPYLOOP1: for (i = 0; i < MATRIX_SIZE; i++)
49     {
50         COPYLOOP2: for (j = 0; j < MATRIX_SIZE; j++)
51         {
52             COV_IN_AUX[i][j] = COV_IN[i][j];
53             EIGENV_AUX[i][j] = EYE[i][j];
54         }
55     }

56     EIGENLOOP: for (h = 0; h < JACOBI_ITERS; h++)
57     {
58         // n/2 rotation angles calculation.
59         xi = 0; yi = 0; xo = 0; yo = 0; zo = 0;
60         ANGLESLLOOP: for(j = 0; j < DIAGONAL_PROCESSORS; j++)
61         {
62             yi = COV_IN_AUX[2*j][2*j+1] << 1;
63             xi = COV_IN_AUX[2*j+1][2*j+1] - COV_IN_AUX[2*j][2*j];
64             cordic_scp(xi, yi, 0, &xo, &yo, &zo, TRANSLATE);
65             rotation_angles[j] = zo >> 1;
66         }

67     }

68

69
70     xo1 = 0; yo1 = 0; zo1 = 0;
71     xo2 = 0; yo2 = 0; zo2 = 0;
72     xo3 = 0; yo3 = 0; zo3 = 0;
73     xo4 = 0; yo4 = 0; zo4 = 0;

74

75     // Nested loops to perform  $S[k+1] = R' * S[k] * R$  operation
76     // over A submatrix.
77     MAINLOOP_1: for (i = 0; i < DIAGONAL_PROCESSORS; i++)
78     {
79         MAINLOOP2: for (j = i; j < DIAGONAL_PROCESSORS; j++)
80         {
81             // First eigenvalues rotation
82             // Saux[k+1] = R' * Saux[k]

83
84             cordic_scp(COV_IN_AUX[2*i][2*j],
85                       COV_IN_AUX[2*i+1][2*j],

```

```

88         rotation_angles[i],
        &xo1, &yo1, &zso1,
        ROTATE);
90
        cordic_scp(COV_IN_AUX[2*i][2*j+1],
92        COV_IN_AUX[2*i+1][2*j+1],
        rotation_angles[i],
94        &xo2, &yo2, &zso2,
        ROTATE);
96
        // Second eigenvalue rotation
98        // S[k+1] = Saux[k+1]*R

        cordic_scp(xo1, xo2, rotation_angles[j],
100        &xo3, // xo3
102        &yo3, // yo3
        &zso3,
104        ROTATE);

        cordic_scp(yo1, yo2, rotation_angles[j],
106        &xo4, // xo4
108        &yo4, // yo4
        &zso4,
110        ROTATE);

112        // Result copy to an auxiliary matrix in the right order.
        COV_IN_TMP[dir[2*i]][dir[2*j]] = xo3;
114        COV_IN_TMP[dir[2*i]][dir[2*j+1]] = yo3;
        COV_IN_TMP[dir[2*i+1]][dir[2*j]] = xo4;
116        COV_IN_TMP[dir[2*i+1]][dir[2*j+1]] = yo4;

118        // If the calculated submatrix result does not correspond
        // to a diagonal one, it's mirrored to make use of initial
120        // matrix symmetry.
        if (i != j)
122        {
            COV_IN_TMP[dir[2*j]][dir[2*i]] = xo3;
124            COV_IN_TMP[dir[2*j]][dir[2*i+1]] = xo4;
            COV_IN_TMP[dir[2*j+1]][dir[2*i]] = yo3;
126            COV_IN_TMP[dir[2*j+1]][dir[2*i+1]] = yo4;
        }
128    }
}

130
// Nested loops to perform eigenvector operation
132 // over the correspondent submatrix
// Vs[k+1] = Vs[k]*R
134 VECTLOOP_1: for (i = 0; i < DIAGONAL_PROCESSORS; i++)
{
136     VECTLOOP2: for (j = 0; j < DIAGONAL_PROCESSORS; j++)
    {
138         // Eigenvector rotation.

140         cordic_scp(EIGENV_AUX[2*i][2*j],

```

```

142         EIGENV_AUX[2*i][2*j+1],
        rotation_angles[j],
        &EIGENV_TMP[dir[2*i]][dir[2*j]],
144         &EIGENV_TMP[dir[2*i]][dir[2*j+1]],
        &Vzo1,
146         ROTATE);

148         cordic_scp(EIGENV_AUX[2*i+1][2*j],
        EIGENV_AUX[2*i+1][2*j+1],
150         rotation_angles[j],
        &EIGENV_TMP[dir[2*i+1]][dir[2*j]],
152         &EIGENV_TMP[dir[2*i+1]][dir[2*j+1]],
        &Vzo2,
154         ROTATE);
    }
156 }

158 // Copy of the reordered data in the initial matrix,
// to setup the next iteration.
160 COPYLOOP3: for (i = 0; i < MATRIX_SIZE; i++){
        COPYLOOP4: for (j = 0; j < MATRIX_SIZE; j++){
162             COV_IN_AUX[i][j] = COV_IN_TMP[i][j];
            EIGENV_AUX[i][j] = EIGENV_TMP[i][j];
164         }
    }
166 }

168 // Result output.
RETURNLOOP1: for(i = 0; i < MATRIX_SIZE; i++){
170     EIGENVALUES[i] = COV_IN_AUX[i][i];
}

172
RETURNLOOP21: for(i = 0; i < MATRIX_SIZE; i++){
174     RETURNLOOP22: for(j = 0; j < MATRIX_SIZE; j++){
        EIGENVECTORS[i][j] = EIGENV_AUX[i][j];
176     }
    }
178 }

```

### C.2.2. cordic\_svd.h

```

1 #ifndef CORDIC_SVD
2 #define CORDIC_SVD

4 #include "cordic_scp.h"

6

8 #define JACOBI_ITERS 22
#define MATRIX_SIZE 8
10 #define DIAGONAL_PROCESSORS 4

12

```

```
14 void cordic_svd( cordic_t COV_IN[MATRIX_SIZE][MATRIX_SIZE],  
                  cordic_t EIGENVALUES[MATRIX_SIZE],  
16                  cordic_t EIGENVECTORS[MATRIX_SIZE][MATRIX_SIZE]);  
    #endif
```

# Bibliografía

- [1] Xilinx, “Vivado design suite user guide: High level synthesis,” 2013, [http://www.xilinx.com/support/documentation/sw\\_manuals/xilinx2013\\_4/ug902-vivado-high-level-synthesis.pdf](http://www.xilinx.com/support/documentation/sw_manuals/xilinx2013_4/ug902-vivado-high-level-synthesis.pdf), Accessed: 2014-02-2.
- [2] C. V. L. Richard P.brent, Franklin T. Luk, “Computation of the generalized singular value decomposition using mesh-connected processors,” *Journal of VLSI and Computer Systems*, vol. 1, no. 3, 1983.
- [3] J. R. Cavallaro and F. T. Luk, “Cordic arithmeric for an svd processor,” *Journal of Paralell and Distributed Computing*, pp. 271 – 290, 1987.
- [4] G. Golub and C. Van Loan, *Matrix Computations*, ser. Johns Hopkins Studies in the Mathematical Sciences. Johns Hopkins University Press, 2013.
- [5] R. P.brent and F. T. Luk, “The solution of singular-value and symmetric eigenvalue problems on multiprocessor arrays,” *Society for industrial and Applied mathematics*, 1985.
- [6] I. Bravo Muñoz, “Arquitectura basada en fpgas para la detección de objetos en movimiento, utilizando visión computacional y técnicas PCA,” Ph.D. dissertation, Departamento de Electrónica de la Universidad de Alcalá de Henares, 2007.
- [7] C. F. Araque, “Diseño de un método eficiente para el cálculo de autovalores y autovectores mediante XSG,” 2010.
- [8] J. E. Volder, “The cordic trigonometric computing technique,” vol. EC-8, no. 3, 1959, pp. 330 – 334.
- [9] J. S. Walther, “A unified algorithm for elementary functions,” 1972.
- [10] E. Antelo, “Algoritmos y arquitectura cordic en aritmética redundante para procesamiento de alta velocidad,” Ph.D. dissertation, Departamento de Electrónica y Computación de la Universidad de Santiago de Compostela, 1995.
- [11] E. W. Weisstein, “Singular value,” 1999, from MathWorld–A Wolfram Web Resource. <http://mathworld.wolfram.com/SingularValue.html>, Accessed: 2014-02-15.
- [12] I. Bravo, P. Jimenez, M. Mazo, J. Lazaro, and A. Gardel, “Implementation in fpgas of jacobí method to solve the eigenvalue and eigenvector problem,” in *Field Programmable Logic and Applications, 2006. FPL '06. International Conference on*, Aug 2006, pp. 1–4.
- [13] Diligent, “Nexys4 fpga board reference manual,” 2013, [https://diligentinc.com/Data/Products/NEXYS4/Nexys4\\_RM\\_VB2\\_Final\\_5.pdf](https://diligentinc.com/Data/Products/NEXYS4/Nexys4_RM_VB2_Final_5.pdf), Accesed: 2014-03-15.

- [14] Y. Liu, C.-S. Bouganis, and P. Cheung, "Hardware architectures for eigenvalue computation of real symmetric matrices," *Computers Digital Techniques, IET*, vol. 3, no. 1, pp. 72–84, January 2009.
- [15] I. Bravo, P. Jimenez, M. Mazo, J. Lazaro, A. Gardel, and M. Marron, "Evaluation and selection of internal parameters of a cordic-unit for a specific application based on fpgas," in *Intelligent Signal Processing, 2007. WISP 2007. IEEE International Symposium on*, Oct 2007, pp. 1–6.
- [16] Xilinx, "Logicore ip cordic v4.0," 2011, [http://www.xilinx.com/support/documentation/ip\\_documentation/cordic\\_ds249.pdf](http://www.xilinx.com/support/documentation/ip_documentation/cordic_ds249.pdf), Accessed: 2014-03-10.
- [17] B. Zhou, R. Brent, and M. Kahn, "Efficient one-sided jacobi algorithms for singular value decomposition and the symmetric eigenproblem," in *Algorithms and Architectures for Parallel Processing, 1995. ICAPP 95. IEEE First ICA/sup 3/PP., IEEE First International Conference on*, vol. 1, Apr 1995, pp. 256–262 vol.1.
- [18] Xilinx, "Vivado design suite tutorial: High level synthesis," 2013, [http://www.xilinx.com/support/documentation/sw\\_manuals/xilinx2013\\_4/ug871-vivado-high-level-synthesis-tutorial.pdf](http://www.xilinx.com/support/documentation/sw_manuals/xilinx2013_4/ug871-vivado-high-level-synthesis-tutorial.pdf), Accessed: 2014-02-2.



Universidad de Alcalá  
Escuela Politécnica Superior



ESCUELA POLITECNICA  
SUPERIOR